Community Edition
Version 0.1.1 | 2026

Front-end Debugging Tools

Chrome DevTools, IDE Debugging, and More...

By Lala Hakobyan
linkedin.com/in/lala-hakobyan

Legal Notice

Disclaimer

The information provided in this handbook is for educational purposes only. It is based on the author's personal experience and research using official documentation and various tools. While every effort has been made to ensure the accuracy and completeness of the content, the world of front-end development is constantly evolving. This guide is not official documentation and should be used as a supplementary resource. The author assumes no responsibility for errors, omissions or any actions taken based on the information provided.

License

Version: 0.1.1 (Community Edition), February 2026

You MAY

- Download, read and use this handbook for learning and reference.
- Print it for personal use or internal training.
- Share the **unmodified** PDF internally, including within your company or educational institution.
- Quote short excerpts with **attribution** and a link to the official source.

You MAY NOT (without prior written permission from the author)

- Re-publish or re-upload this handbook publicly (website, platform or another repository).
- Sell the handbook or include it in paid products/courses.
- Distribute modified versions of the handbook text or images.

Code Snippets
Original code snippets authored by Lala Hakobyan are licensed under the MIT License and may be used in personal and commercial projects.

Third-party Content
Any third-party code/assets that are credited should remain under their original licenses.

Official Source
For the latest version and new updates, please refer to the official repository:
https://github.com/lala-hakobyan/front-end-debugging-handbook

Feedback and Requests
For suggestions, feedback or to request a topic, please use the Issues tab on GitHub:
https://github.com/lala-hakobyan/front-end-debugging-handbook/issues

Table of Contents

Introduction

In an era where AI can generate code in seconds, the skill of **debugging** has become more critical than ever. AI often acts like a smart junior developer who is knowledgeable but lacks real-world experience. Our role as engineers is to **validate**, **debug** and **harden** that code for production.

During my career, I witnessed even strong senior engineers relying primarily on the **Console** for debugging. While `console.log()` is a fast and useful tool (don't get me wrong: I also use it for quick debugging), it's often insufficient for debugging today's complex, multi-layered front-end applications with their distinct UI, state management and data layers. Furthermore, when you encounter an urgent production issue, you don't have the time to add console logs, redeploy the application and debug that version with logs. You need to already have powerful tools at your fingertips to act quickly and keep customers satisfied.

In this guide, I will share the set of powerful tools I recommend and use personally in my debugging workflows. This is my personal toolkit for gaining the deep visibility needed to debug complex, modern applications efficiently. It combines my 9 years of experience in front-end engineering with new and modern tools I learned and adopted during this past year.
You will learn **how to start with a tool**, **dive deep into the concepts** associated with it and **understand when to use it**. Whether you debug a local application, AI-generated code, production bugs, or just perform quality testing, the knowledge gained in this handbook will help make your debugging process smoother and faster, saving you countless hours.

To verify the concepts in this handbook, **I relied only on authoritative resources**: official documentation from the Google team, deep dives by community experts, as well as framework and library teams (React, Angular, Redux). At the end of the handbook, you will find the Documentation and Resources chapter to continue and dive deeper.

All screenshots in this handbook are from my personal POC projects using **recent versions of the technologies** (Angular 20, Next.js 16, React 19, Node.js 22). I also used the recent versions of the Chrome browser, IDEs (Cursor, WebStorm) and extensions (Redux DevTools, React DevTools and Angular DevTools) to present the most up-to-date information for you. You can find more about the projects and tools used in the screenshots in a separate Technical Configuration chapter at the end of the handbook.

It is important to note that the purpose of this handbook is not to replace official documentation. Instead, it is aligned with it and, in some cases, offers more practical, up-to-date guidance. The goal is to empower you with a comprehensive toolset to debug complex front-end applications in the era of AI. While I plan to release updates as tools evolve, technologies move very fast. New features can be added or removed, and interfaces can be updated, but the core toolset and debugging principles will remain the same.

I hope you enjoy this handbook and find it genuinely useful in your everyday debugging flows.

1. Chrome DevTools

Chrome DevTools is a set of web developer tools built directly into the Google Chrome browser used for **inspecting**, **debugging** and **optimizing** web pages and applications. It allows developers to edit pages on the fly, diagnose problems, analyze network activity and test performance in real-time.

You can open the developer tools in Google Chrome in a few common ways:

- **Right-click Menu**: Right-click anywhere on a webpage and select **Inspect**. This is often the easiest way to open the tools and directly inspect a specific element.
- **Keyboard Shortcuts**:
  - **Windows or Linux**: `F12` or `Ctrl+Shift+I`
  - **macOS**: `Cmd+Option+I`

After opening the developer tools, you can use the **Command Menu** to open the desired panel.

- Open the Command Menu using these shortcuts:
  - **Windows or Linux:** `Ctrl+Shift+P`
  - **macOS:** `Cmd+Shift+P`
- After the Command Menu is opened, type the desired panel name and click on it to open it. Typically, panel names are shown with the **Show** prefix in the autocomplete list.

1.1 Console Panel

The **Console** is often the first stop for any debugging session. It provides a live, interactive JavaScript environment and a running log of messages from your application.

While `console.log()` is the most common method, the `console` object has several other methods that are extremely useful for providing clearer and more contextual output. However, relying only on console logging for debugging can be inefficient and may **slow down your debugging workflow**. It is advisable to use a combination of different debugging techniques to achieve maximum efficiency.

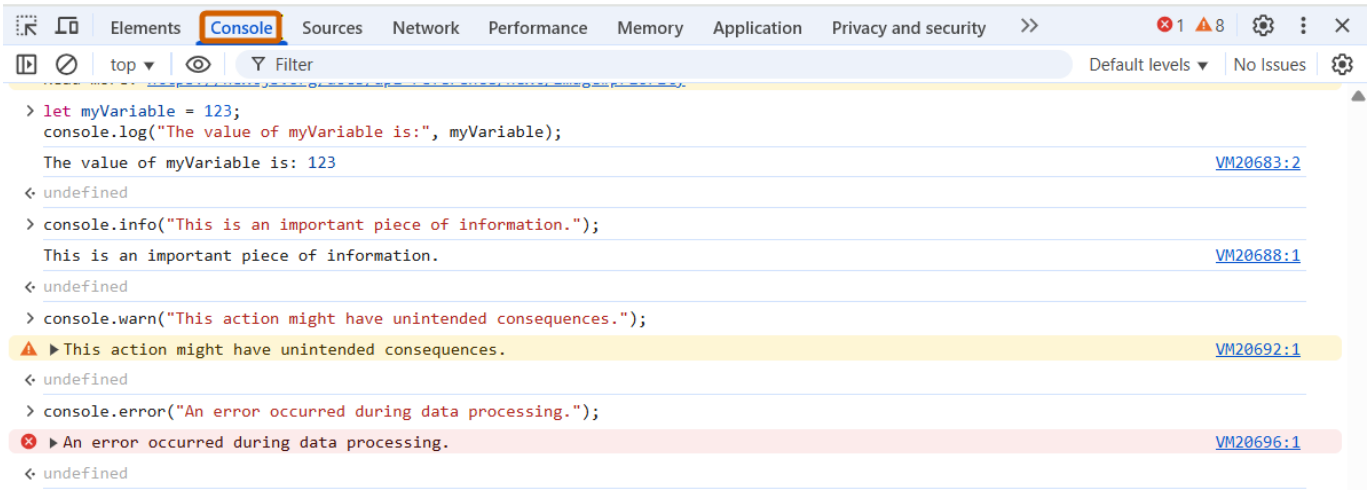
Console Panel: How to Open

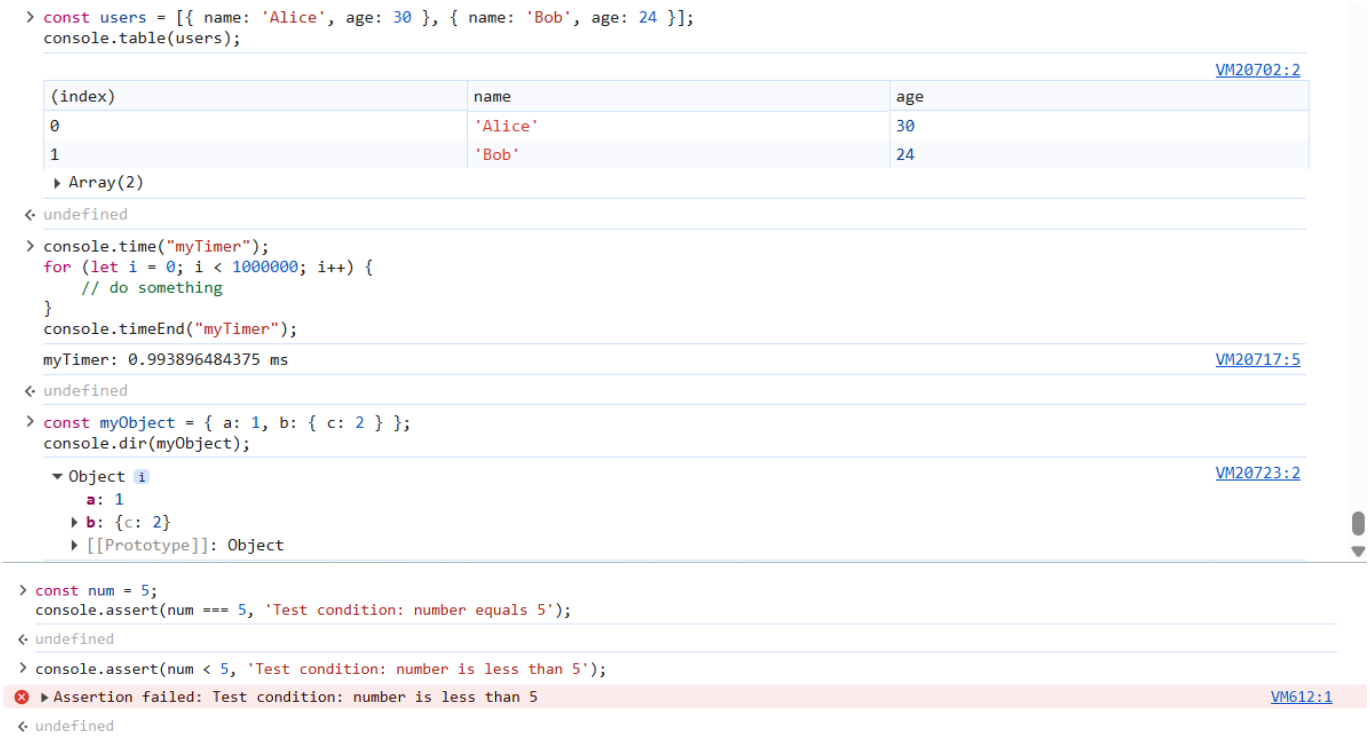
To open the **Console** panel, open DevTools and click the **Console** tab, or select it from the **Command Menu**.

Console Panel: Useful Console Commands

- **`console.log()`**: The standard method for general output and messages.
- **`console.info()`**: Used for informational messages, often styled with a blue "i" icon.
- **`console.warn()`**: Outputs a warning message, typically in yellow, indicating a potential issue.
- **`console.error()`**: Outputs an error message, typically in red, and often includes a stack trace.
- **`console.dir(object)`**: Displays an interactive, expandable tree view of a JavaScript object's properties.
- **`console.table(arrayOrObject)`**: Renders an array of objects (or an object) as a clean, sortable table, which is much easier to read than a standard log.
- **`console.time('label')`** and **`console.timeEnd('label')`**: Used for measuring duration. Starts a high-precision timer with a specific label and later stops it, logging the elapsed time in milliseconds to the **Console**.
- **`console.assert(condition, message)`**: Outputs the `message` as an error only if the `condition` is `false`. If the `condition` is `true`, it does nothing. This is very useful for writing simple, inline test assert commands without cluttering the **Console** with success messages.

Below is an example demonstrating these commands in action in the **Console** panel:





Console Panel: Useful Features

The **Console** is an interactive environment where you can execute JavaScript code. It also includes several powerful utility functions for easier manipulation of DOM elements as part of the **Console Utilities API**:

1. `$0, $1, $2, $3, $4`

- In the **Elements** panel, select a few different DOM elements. You can then switch to the **Console** panel and use `$0` to reference the most recently selected element, `$1` for the one selected before that, and so on up to `$4`.
- Example:

```
$0.style.backgroundColor = 'red';
$1.remove();
$2.textContent = 'Updated text';
```

2. `$()`

- This **Console utility function** is a shortcut for `document.querySelector()`, providing a quick way to select elements on a web page using CSS selectors. It returns the first element that matches a CSS selector.
- Example:

```
$('h1').textContent;
```

3. `$$()`

- Similarly, this **Console utility function** is a shortcut for `document.querySelectorAll()`. It returns an array of all elements that match a CSS selector. The main advantage is that unlike the original method, which returns a `NodeList` (array-like list), the shortcut returns a **JavaScript array** which allows you to immediately use array methods like `.map()` or `.filter()`.

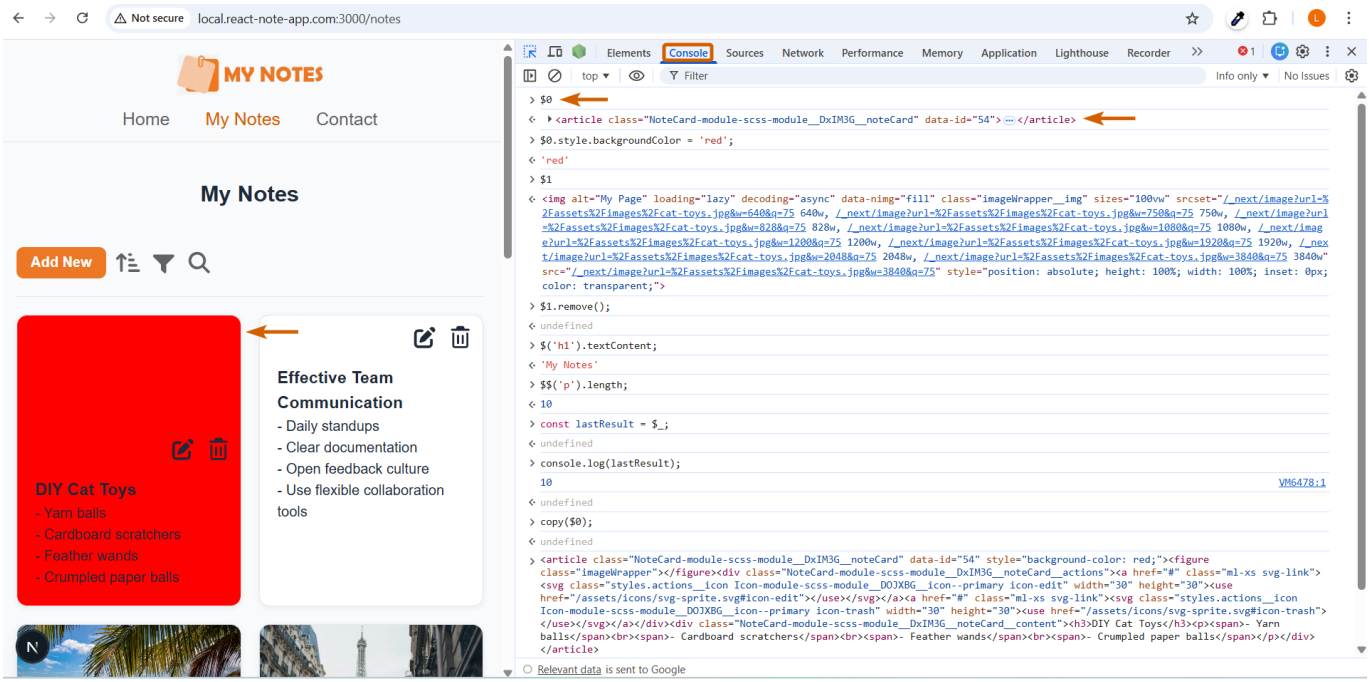
```
$$('p').length;
```

4. `$_`

- Refers to the result of the **last evaluated expression in the Console**.
- Example: Do the following steps in the Console:
  - Write this expression: `{id: 1, type: 'fruit', name: 'apple'}`
  - Declare the `lastResult` variable and assign `$_` to it: `const lastResult = $_;`
  - Log the `lastResult` value: `console.log(lastResult);`
  - You will see that it prints `{id: 1, type: 'fruit', name: 'apple'}` because `lastResult` stores the last evaluated expression.

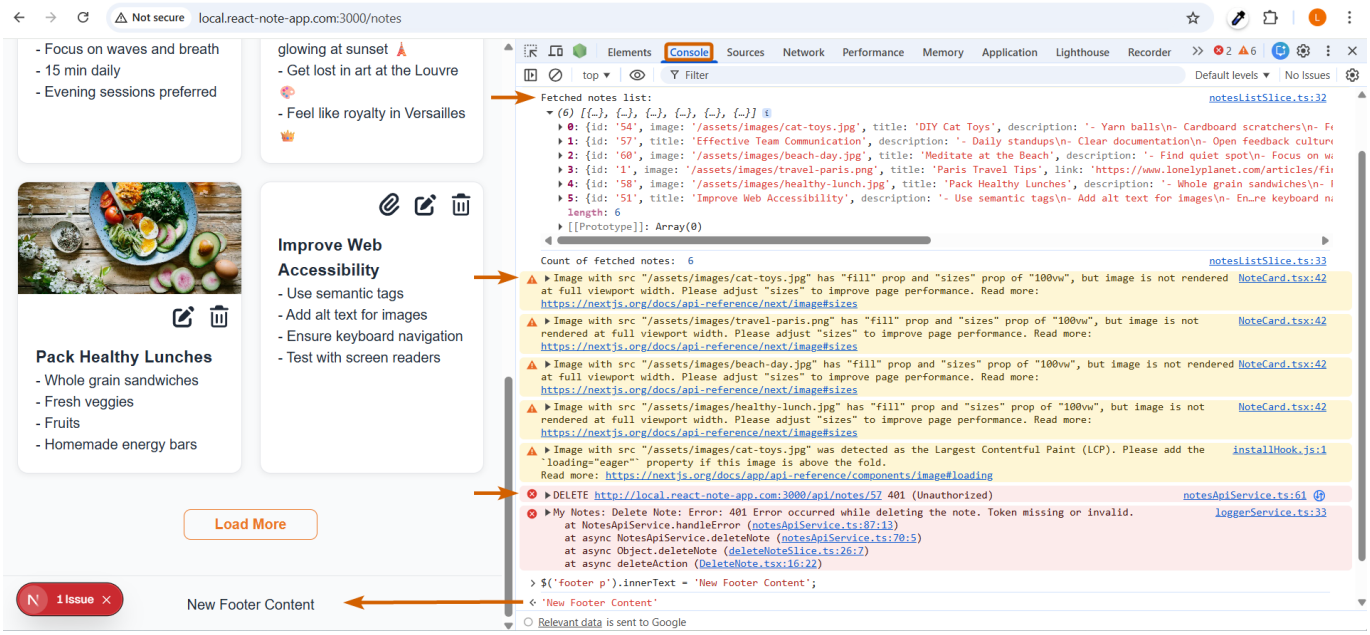
5. `copy()`

- **Copies a JavaScript value** (string, object, DOM node) to your system clipboard.
- Example:
  - Run `copy({ name: 'John', age: 30 });` or `copy($0)` in the Console.
  - You can now paste the copied content anywhere, including back into the Console or into a text editor.



Console Panel: Common Use Cases

- **Immediate Error Overview**: Getting a high-level summary of all errors and warnings as soon as you open the page.
- **Quick Debugging and Value Inspection**: Using methods like `console.log()`, `console.table()` and `console.dir()` to quickly check the value of a variable at a specific point in time.
- **Live Prototyping and Testing**: Running JavaScript code directly or using **Console Utility functions** (e.g., `$0`, `$()`, `$$()` etc.) in the browser to test a function, change CSS styles, or manipulate the DOM without having to modify your source code and reload the page.



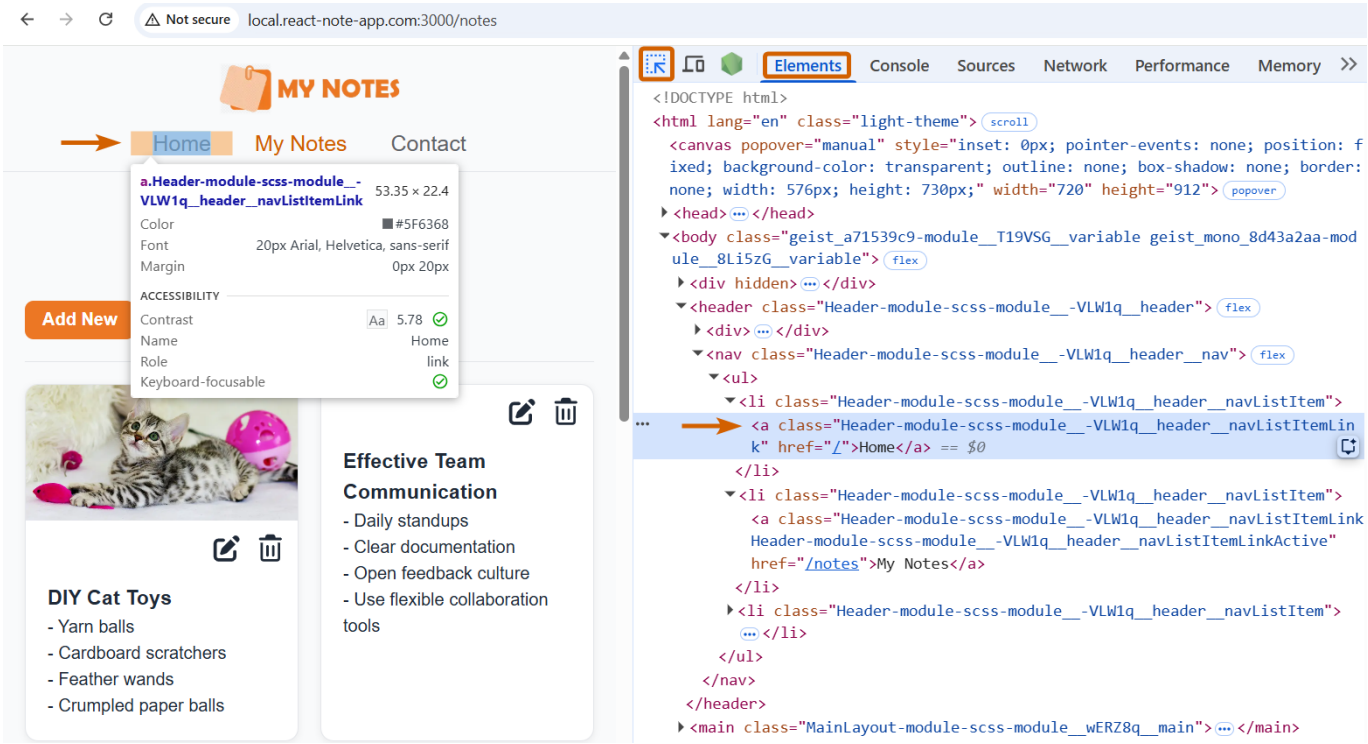
1.2 Elements Panel

The **Elements** panel provides a live, interactive interface for inspecting and manipulating the **Document Object Model (DOM)** and its associated **CSS styles**. It renders the page's HTML as a navigable tree, allowing you to select any node and see exactly how it's styled and structured.

Elements Panel: How to Open

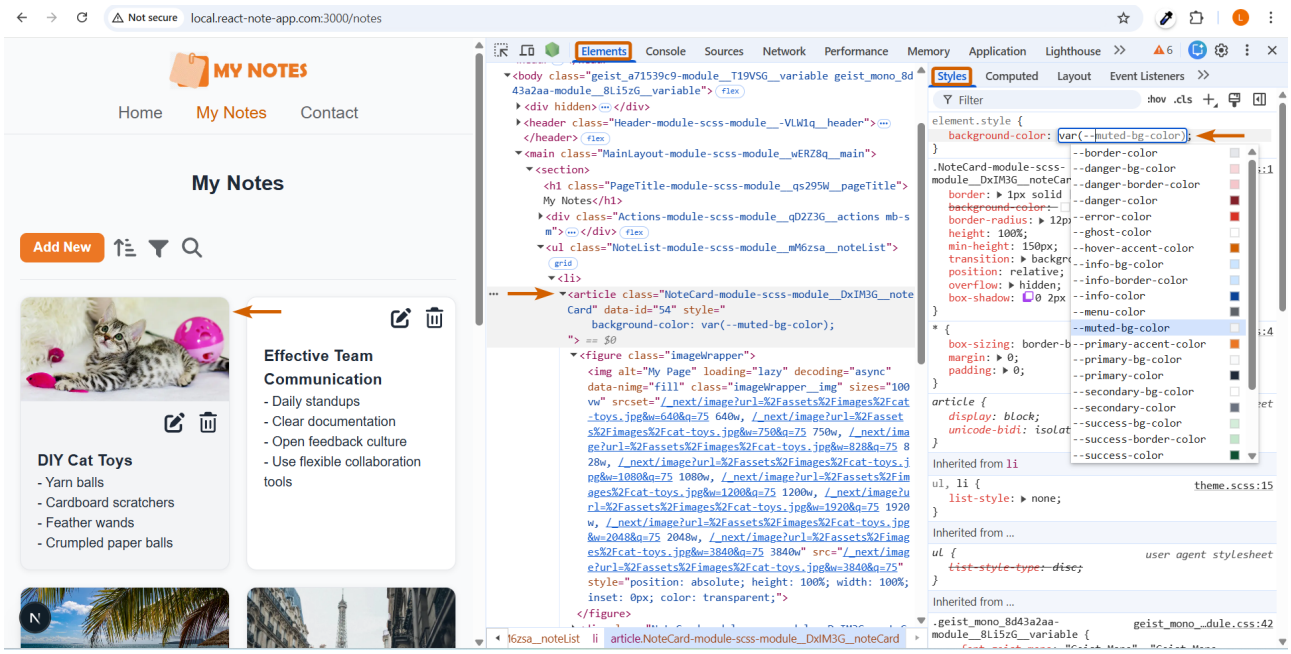
After opening DevTools, you can inspect an element in the **Elements** panel in two main ways:

- Right-click the element on the page and select **Inspect**.
- Click the selector (↖) icon in the top-left corner to activate the element **selector tool**.



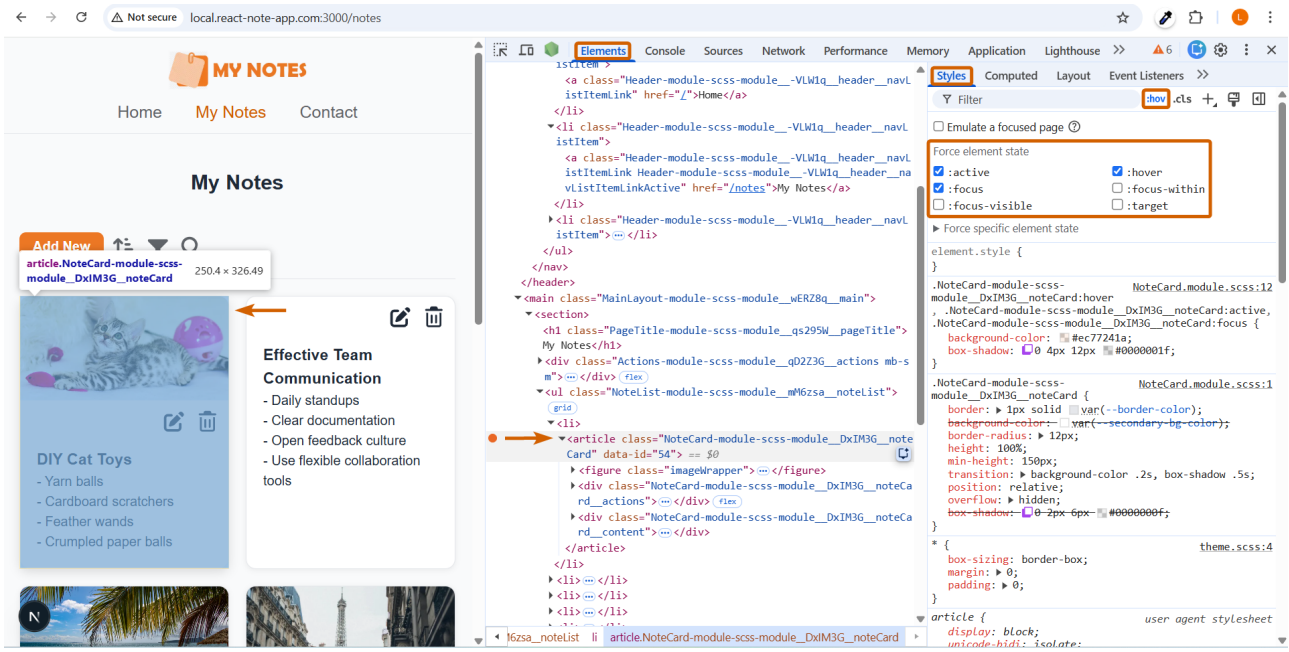
Elements Panel: Common Use Cases

- **Live CSS Editing and Prototyping**: The primary use case of this panel is debugging styles. You can view all CSS rules applied to an element in the **Styles** tab, see the final computed values in the **Computed** tab and add or modify styles directly in the browser. This is useful for rapidly prototyping design changes before writing any code.

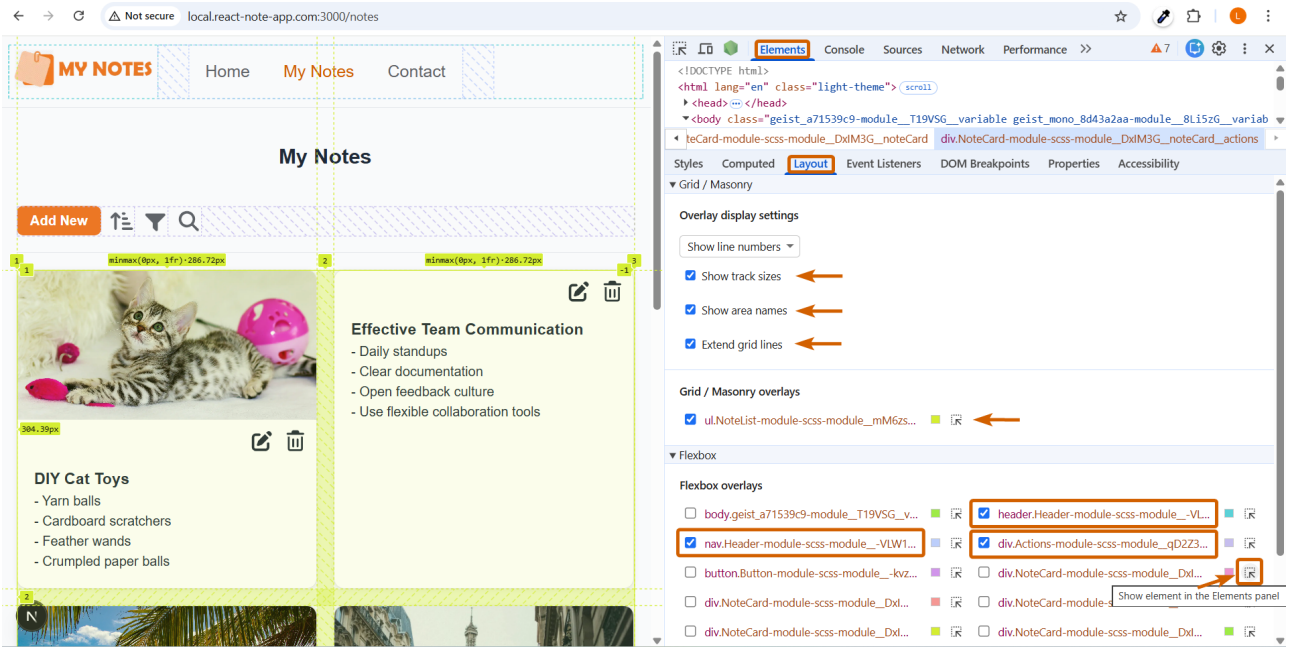


- **DOM Manipulation**: Directly edit the HTML by adding, removing or reordering DOM nodes in the tree. You can also add or remove CSS classes to see how your component's styling responds.

- **Inspecting Element States**: Force elements into states like `:hover`, `:active` or `:focus` by navigating to **Styles > :hov** section. This allows you to debug element-specific states right from DevTools without having to manually interact with them to force the state on the page.



- **Debugging Layouts**: Check for issues with page layout, spacing and alignment from the **Layout** tab. This tab is essential for **pixel-perfect** design checks and is useful for debugging CSS Grid and Flexbox overlays. You can also use the **Show element in the Elements panel** (↖) icon next to the component to inspect the specific component on the page and highlight it in the Elements panel.



- **Auditing Accessibility**: Inspect the accessibility tree from the **Accessibility** tab to see how **assistive technologies** (tools for users with disabilities) interpret your page. You can check for missing `aria-` attributes, incorrect roles and other common accessibility issues.

Note: If you need to debug CSS animations, CSS transitions, web animations or the View Transitions API, you can use the **Animations** panel in DevTools.

How to open it:

- Open DevTools and open the **Command Menu** by pressing:
  - **Windows or Linux**: `Ctrl+Shift+P`
  - **macOS**: `Cmd+Shift+P`
- Type **Show Animations** and select the panel from the list.

The **Animations** panel allows you to slow down, replay and inspect animations, as well as modify their timing, delay, duration and keyframe offsets.

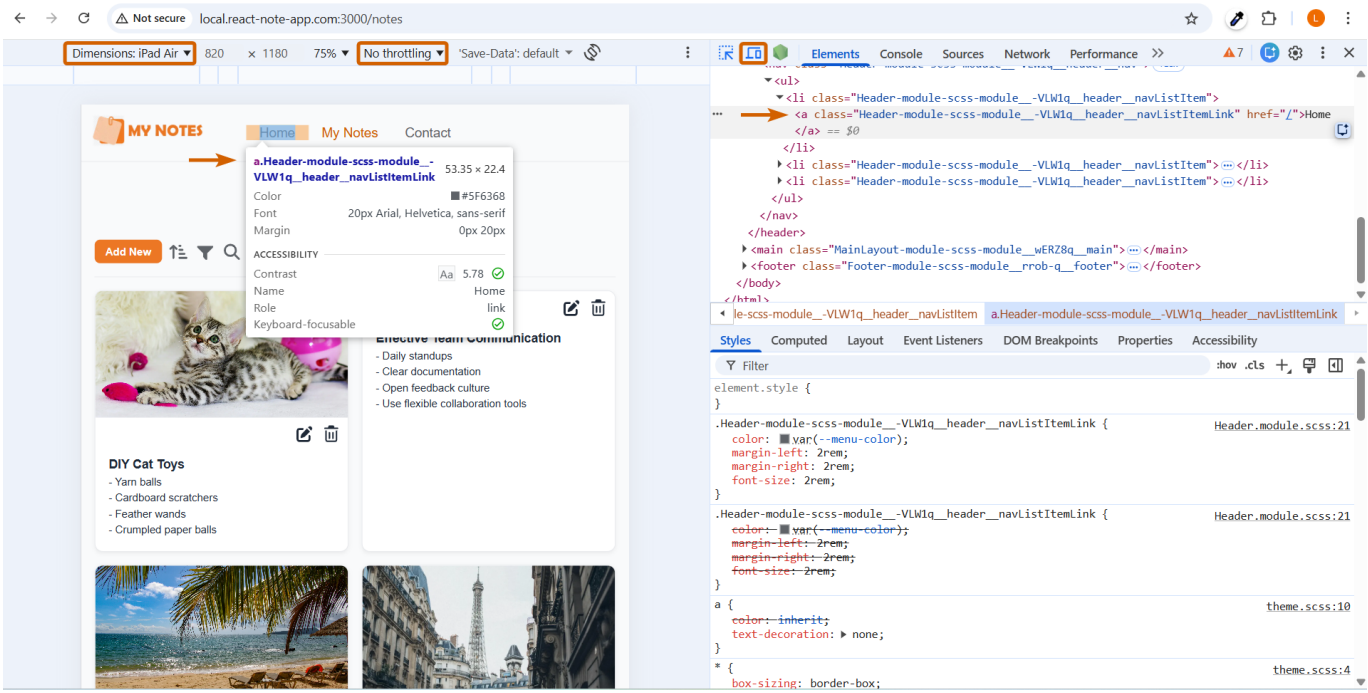
1.3 Toggle Device Toolbar

The **Toggle Device Toolbar** is a feature within Chrome DevTools that allows you to simulate how your web application looks and performs on various mobile devices and screen sizes. It's essential for testing responsive layouts and ensuring a consistent user experience across different viewports.

Toggle Device Toolbar: How to Activate

After opening DevTools, you can activate the **Toggle Device Toolbar** in two main ways:

- Click the **Toggle device toolbar** icon (it looks like a phone and tablet) on the left of the DevTools **top action bar**.
- Use the keyboard shortcut: `Ctrl+Shift+M` (Windows/Linux) or `Cmd+Shift+M` (macOS).



Toggle Device Toolbar: Key Features

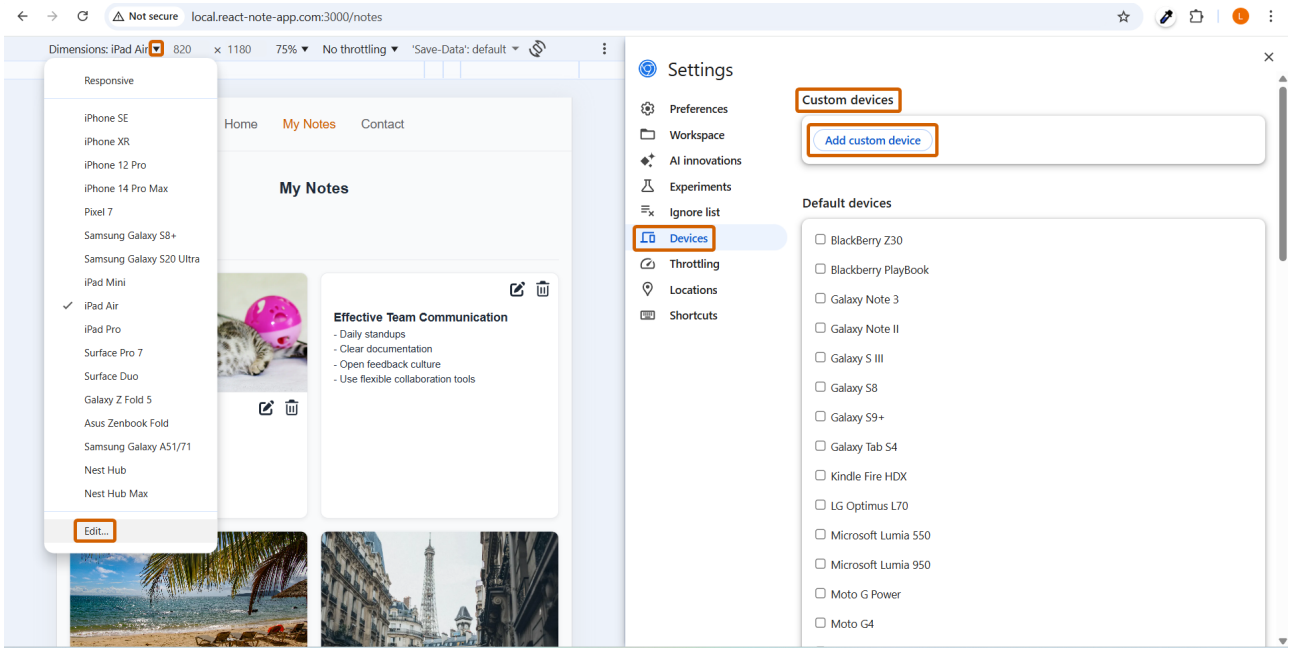
Once active, the toolbar provides several powerful features:

- **Responsive Testing**: Instantly see how your layout adapts to different screen dimensions without resizing your browser window.
- **Device Presets**: Choose from a wide range of popular devices like iPhones, Samsung Galaxy phones, iPads and Google Pixels to test on their exact viewport dimensions.
- **Network Throttling**: Simulate different mobile network speeds to test your application's performance under real-world conditions.
- **Custom Viewports**: If a device isn't on the list, you can manually enter any width and height to test specific or unusual breakpoints.

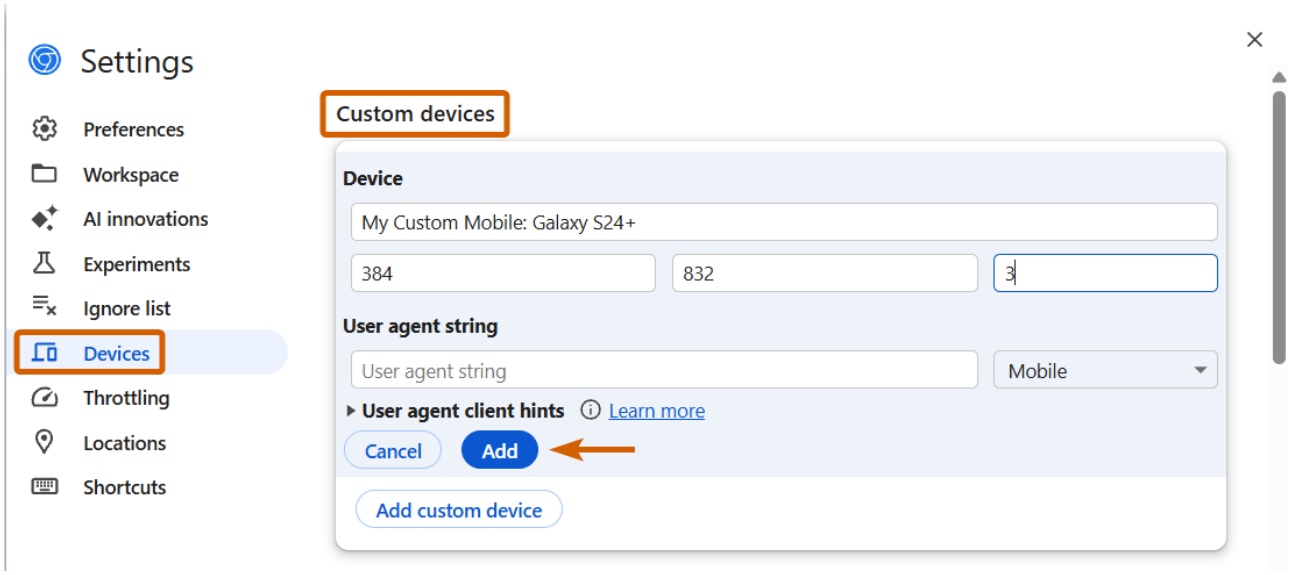
Toggle Device Toolbar: How to Add a Custom Device

If you need to test a specific device that isn't in the default list, you can easily add your own custom profile.

1. With the Toggle Device Toolbar active, click the dropdown menu at the top of the viewport that shows the currently selected option (e.g., **Responsive**, **iPad Air**).
2. At the bottom of this dropdown list, click the **Edit...** option.
3. This opens a settings panel listing all emulated devices. From here, click the **Add custom device...** button.



4. Fill in the form with the device's specifications:
   - **Device Name**: (e.g., `My Custom Mobile: Galaxy S24+`)
   - **Width**: The viewport width in pixels (e.g., `384`)
   - **Height**: The viewport height in pixels (e.g., `832`)
   - **Device pixel ratio**: (e.g., `3` for modern mobile devices, `2` for tablets and retina displays)
5. Click the **Add** button. Your new device will appear in the device list for you to select at any time.



Toggle Device Toolbar: Common Use Cases

- **Resolution Testing**: Debug and test the functionality of your applications in different device resolutions, including your own custom viewports.
- **Touch Event Simulation**: When the device toolbar is active, your mouse cursor simulates touch events, allowing you to test functionality designed for touch-enabled devices.
- **Pixel Perfect Testing**: Perform **pixel-perfect** testing on various devices to ensure the final view is matched with the provided UI/UX design file (like Figma).

1.4 Performance Panel

The **Performance** panel is the primary tool for **performance profiling**, which is the process of analyzing your application's runtime performance, including code execution, rendering and memory usage. It allows you to record what happens as your page loads and runs, so you can measure and improve performance.

> **Note:** The meaning of terms marked with * can be checked in the Performance Panel: Terminology section.

Performance Panel: How to Open

To open the **Performance** panel, open DevTools and click the **Performance** tab, or select it from the **Command Menu**.

Performance Panel: Setup Best Practices

Development servers for modern frameworks add additional overhead such as for supporting *HMR (Hot Module Replacement)** . Some frameworks and libraries call the same function two times to help you surface mistakes at early stages (e.g., during local development in **Strict Mode**, **React** calls each component's function twice, which can help surface mistakes caused by impure functions). Moreover, you may have a lot of extensions installed in the browser which may interfere with performance results (e.g., Redux DevTools, React DevTools, Angular DevTools).

So, it is important to follow these best practices to get accurate performance results:

- **Use production builds:** If the framework or library supports it, the better way is to record performance profile on top of production build. For example, in Next.js, you can use `--profile` option with your production build and it will enable custom tracks and framework-specific data.
- **Use an Incognito window:** Chrome extensions can interfere with performance recordings. Profiling in an Incognito window (or a dedicated Chrome profile) ensures that extensions don't skew your results.
- **Simulate real user conditions:** Your development machine is likely much faster than most users' devices. Use **CPU and Network throttling** to get a more realistic picture of how your application performs in the real world.

Performance Panel: Terminology

Below are short explanations for terms that you will encounter across this section, marked with *:

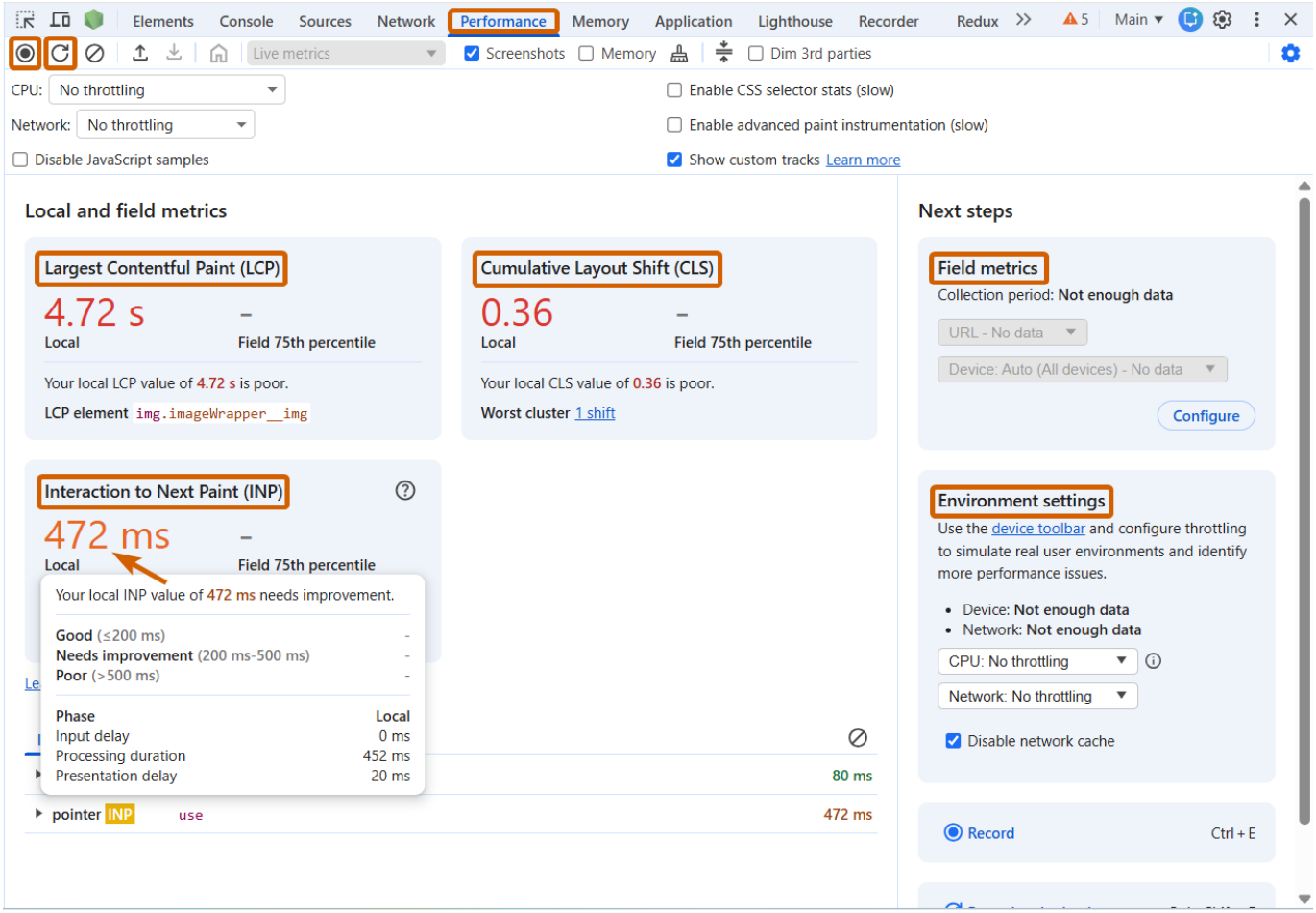
- **HMR (Hot Module Replacement):** A feature that exchanges, adds, or removes modules while an application is running, without a full page reload. Its main advantage is that the browser is instantly updated when modifications are made to CSS/JS files in the source code within your IDE, while

preserving your application state. Therefore, it is used in modern frameworks (e.g., Angular, Next.js) to make local development and debugging processes faster and more effective.

- **Drop-off rate:** Analytics metric that measures the percentage of users who exit a multistep process on a web application after starting it.
- **Bounce rate:** Analytics metric that measures the percentage of visitors who view only one page on a web application and then leave without interacting further.
- **Frame rate: FPS (frames per second):** A measure of motion fluidity that indicates how many still images (frames) are displayed on the screen each second. It represents the speed at which the browser is able to recalculate, layout and paint content to the display.
    - Higher FPS results in smoother motion, while lower FPS makes the movement appear janky.
    - The goal frame rate for the web is **60 FPS**, which aligns with the `60 Hz` refresh rate of most screens, ensuring that animations look smooth to the human eye.
    - If the frame count per `1` second (`1000 ms`) is limited to `60`, each non-idle frame should be rendered in a maximum of about **1000 / 60 ≈ 16.7 ms**.
    - If rendering takes longer than `16.7 ms`, the browser cannot render the next frame appropriately, and this may result in a **dropped frame**.
- **Long Task:** A **task** on the main thread taking over `50 ms`. These tasks can make the page unresponsive to user input, even if it looks ready.
- **Task:** Any discrete unit of work performed by the browser, including JavaScript execution, HTML/CSS parsing, rendering and other internal processes.
- **Rasterization:** A separate step in the browser rendering pipeline responsible for converting the drawings into actual pixel bitmaps.
  In modern browsers, rasterization often happens with GPU assistance to achieve high frame rates and offload work from the CPU.
- **Root activities:** Those activities that cause the browser to do some work. For example, when you click on a page, the browser fires an `Event` activity as the root activity. That `Event` then might cause a handler to execute. Other common root activities include `Paint`, `Commit` and `Layerize`.
- **JS (JavaScript) Heap:** A large, unstructured region of computer memory used for dynamic memory allocation in the JavaScript runtime environment. Unlike stack memory, which has a fixed size and follows a last-in, first-out (LIFO) order, heap memory allows for more flexible allocation and deallocation of memory blocks during runtime. This is where JavaScript stores complex data structures, such as objects, arrays and functions.
- **Memory Leak:** Occurs when an application allocates memory on the heap but fails to deallocate it after use. This unreleased memory remains reserved, reducing the amount of available memory for the rest of the application and other applications. Memory leaks can lead to significant performance degradation, or even cause an application to crash if the system runs out of available memory.

Performance Panel: Key Features and Workflow

Before you even start recording, the Performance panel displays initial metrics and configuration options.



This initial view allows you to:

- **Review Core Web Vitals**:
  The panel immediately shows scores for three key metrics from your local session:

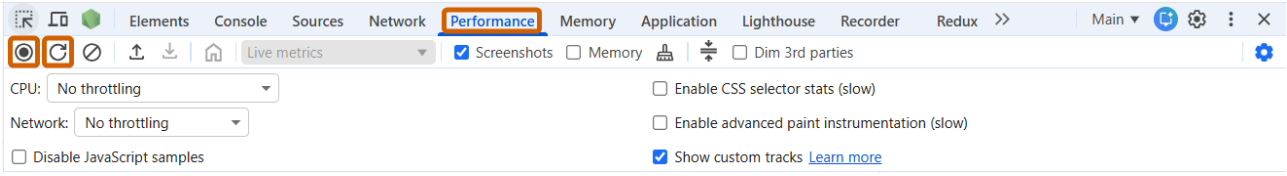
  - **Largest Contentful Paint (LCP)**: Measures **loading performance** by timing how long it takes for the largest content element to become visible.
    (**Good:** <= `2.5 s`, **Needs Improvement:** `2.5 - 4.0 s`, **Poor:** > `4.0 s`)
    If LCP is high, it will directly affect the loading experience of your application. Users will feel that the page is slow and the ***bounce rate**** will increase.
    It is **crucial to preload LCP resources with high priority** to ensure a smooth user experience and a low LCP (e.g., by using `<link rel="preload">` and the `fetchpriority` attribute).

  - **Cumulative Layout Shift (CLS)**: Measures **visual stability** by scoring how much the page content unexpectedly shifts during loading.
    (**Good:** <= `0.1`, **Needs Improvement:** `0.1 - 0.25`, **Poor:** > `0.25`)
    Basically, it measures how much unexpected layout movement occurs on the page, where elements shift in ways that are not caused by a user action (e.g., clicking a button) or by an animation.
    Layout shifts may be caused, for example, by `<img>` or `<video>` elements that do not have initial width and height attributes, so the browser does not know how much space they will occupy until they load.
    If CLS is high, users will likely experience unpleasant interactions such as clicking on something

they did not mean to click or being unable to use important features that the application provides, and therefore ***drop-off rate**** will increase.
It is important to keep this metric minimal by using **predefined width and height for images and videos, and by showing skeleton loaders, placeholders or blurred thumbnails while the content is loading**.

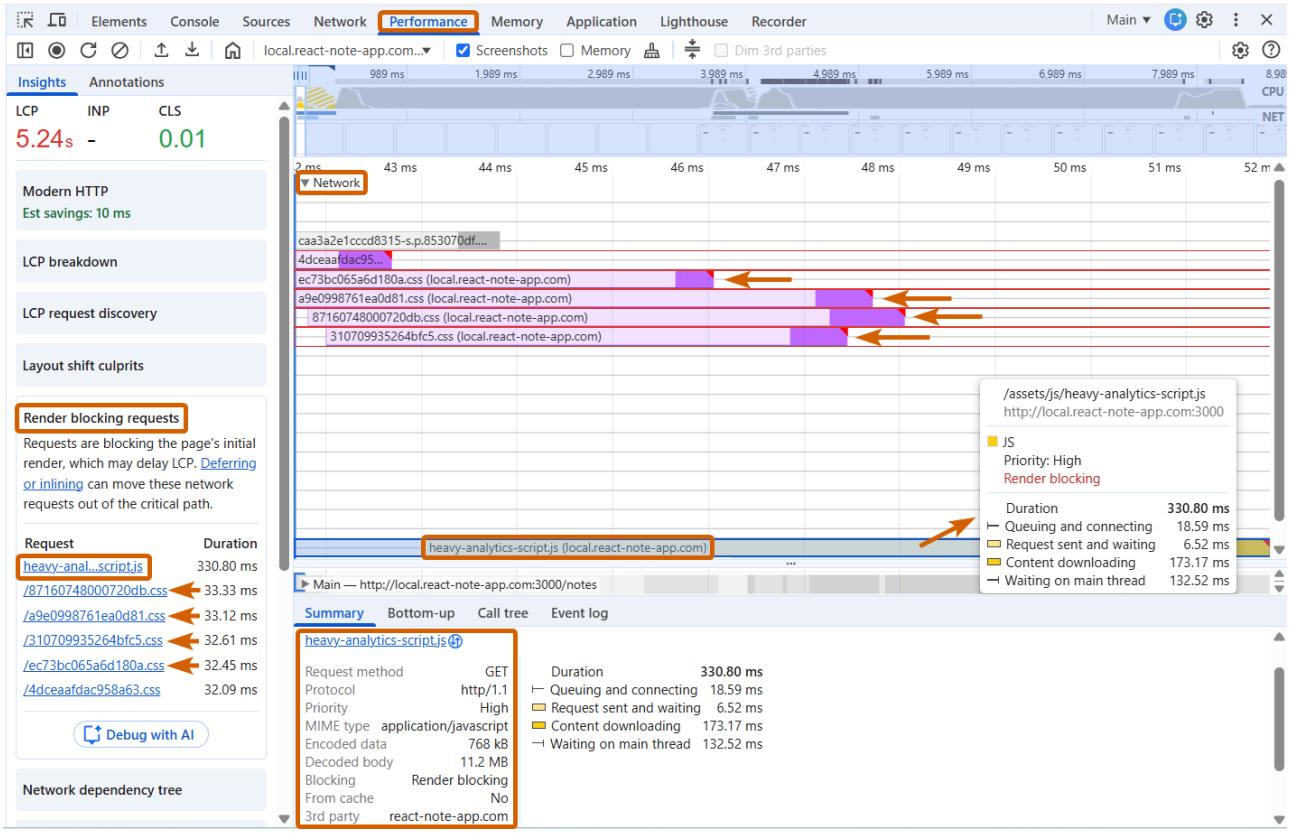
- **Interaction to Next Paint (INP)**: Measures **responsiveness** by tracking the time from a user interaction to the next visual update.
(**Good:** `<= 200 ms`, **Needs Improvement:** `200 - 500 ms`, **Poor:** `> 500 ms`)
If INP is high, it means that users experience slow interactions. For example, they may have to wait noticeably long for a modal to open or for an added item to appear in a list, which will significantly increase the **drop-off rate** of your application.
In these cases, it is advisable to **show a loading spinner or any other indicator to improve user experience** so the user understands what they are waiting for.

- **Simulate Real-User Environments**: From the **Environment settings**, you can apply Network and CPU throttling to mimic how your application performs on slower devices and connections.
- **Compare with Field Data**: The **Field Metrics** section allows you to connect to real user data (if available) via a production URL. This lets you compare your local recording against what actual users experience.
- **Record Performance Profile**
  - **Record Interactions**: To capture a performance profile for a specific scenario, use the **Record** (●) icon to start the recording, perform specific actions to reproduce the desired scenario and click the **Stop** (■) icon to finish the recording.
  - **Record Page Load**: To capture a performance profile for the initial load performance, use the dedicated **Start profiling and reload page** (⟳) icon to start the recording. You can either use the **Stop** (■) icon to finish the recording or wait until the recording finishes automatically, capturing all details of page loading.



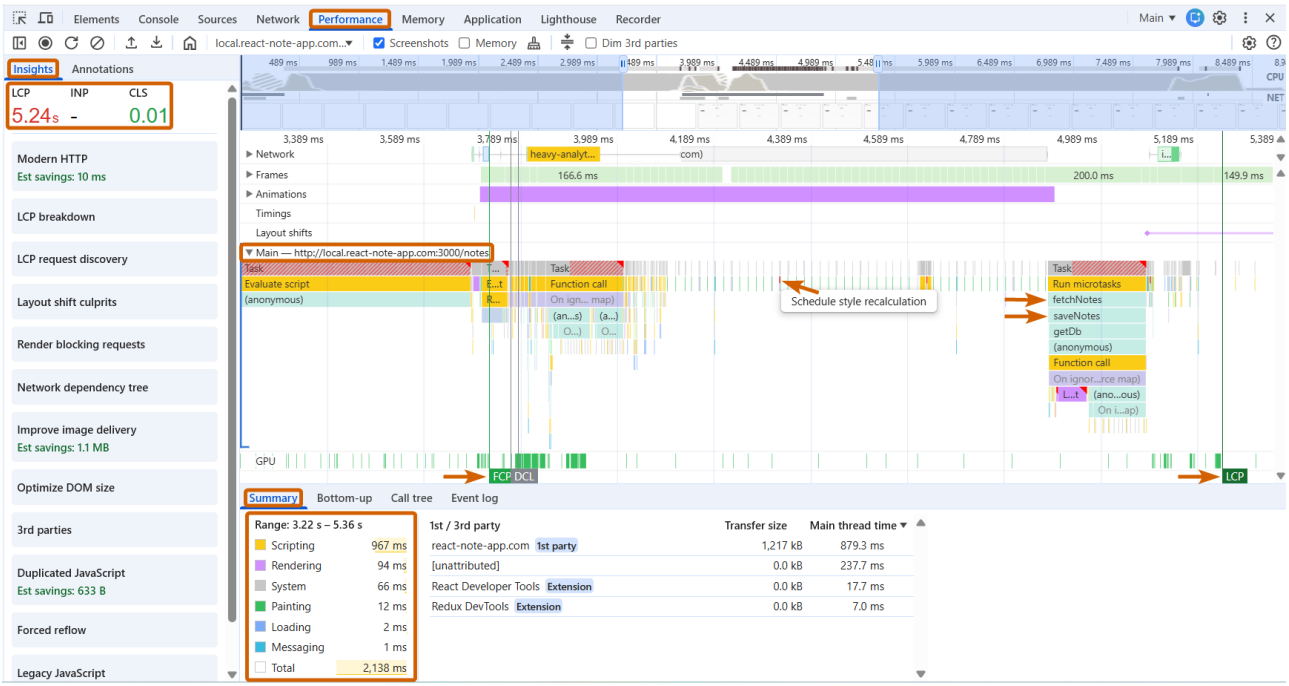
Once a recording is finished, you can use several features to analyze the performance trace of your application:

- **Check Render-blocking Requests**: A good workflow is to start by checking for any **render-blocking requests** in the **Insights** panel. This can help you quickly identify potential bottlenecks.
Generally, the **Insights** panel in the left sidebar is a good starting point for exploration because, besides **render-blocking requests**, it provides actionable insights on your application's performance overall.



- **Correlate Timelines Visually**: It's highly effective to visually connect the **Screenshots** with the **Network** and **Main** tracks. For example, this allows you to identify what was happening in the JavaScript main thread and network while a specific action was occurring in the UI.

- **Review the Summary**: Check the **Summary** tab to see a breakdown of time spent on different activities like **Scripting**, **Rendering**, **Painting** and **System** for the entire recording or for a selected time range.

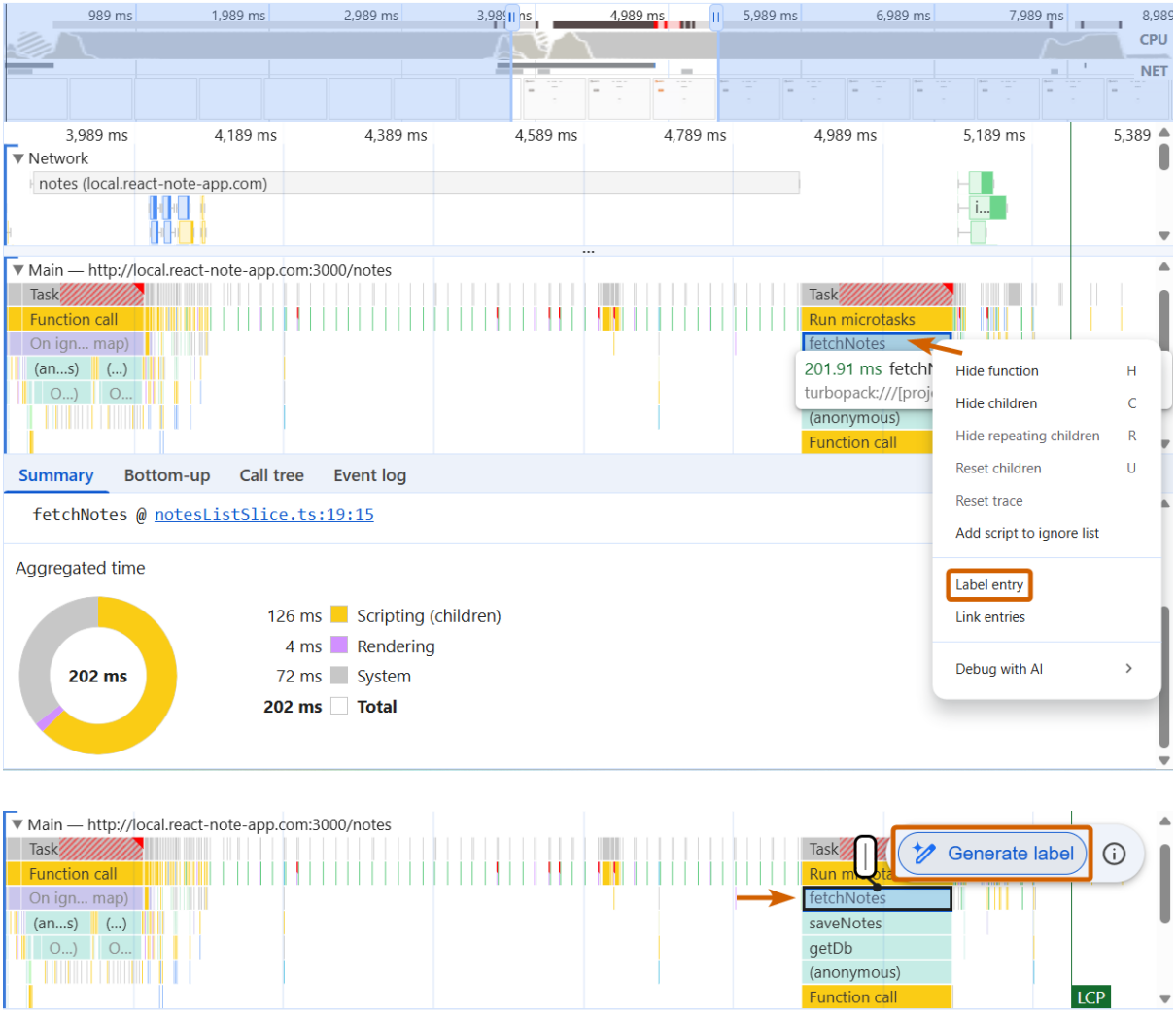


  - Expand the **Main** section if you want to see a breakdown of all activities on the Main thread or alternatively select any section that you need to see the breakdown for.

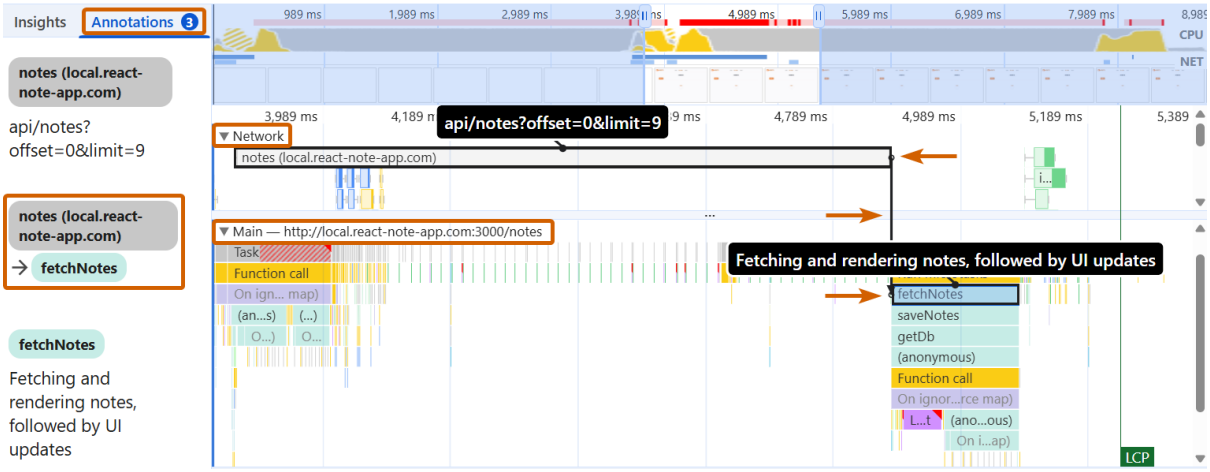
- If you also click on **LCP**, **DCL**, **FCP** timeline markers, you will be able to view the summary for each specific event such as which element caused the LCP (Largest Contentful Paint) or how long FCP (First Contentful Paint) took.

- **Use Annotations and Share Your Findings**: After finding problematic areas, you can mark them using the **Annotations** feature.
  There are several ways to add an annotation:

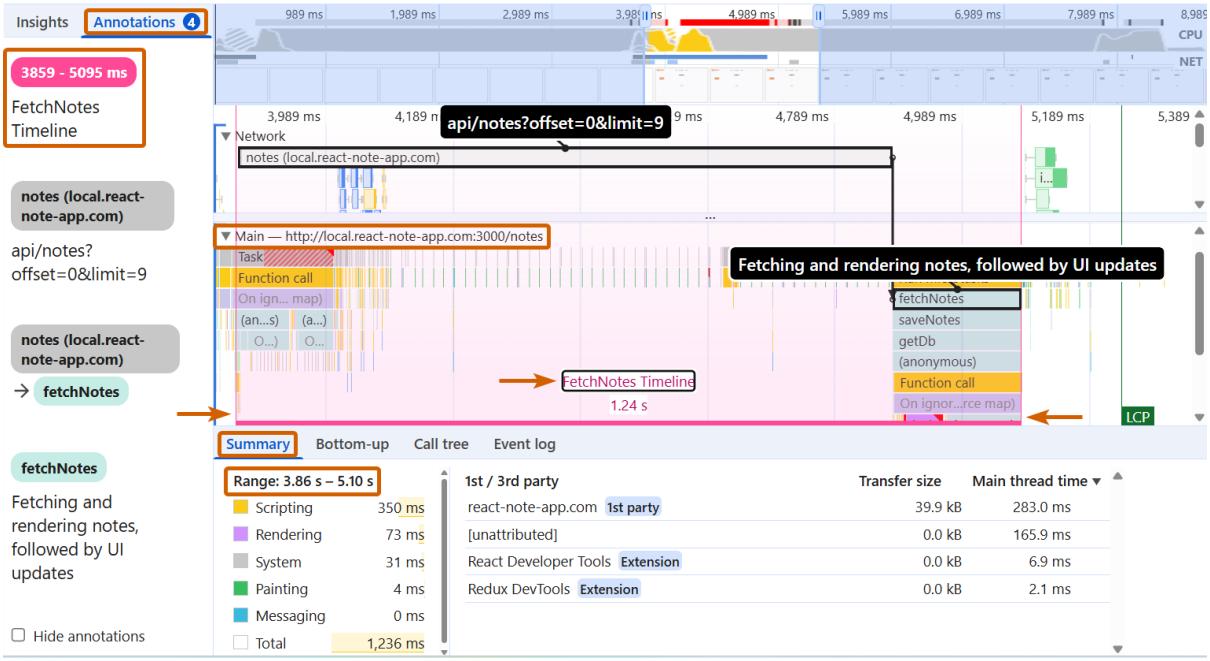
  - **Label item:** To add a custom label to an item, double-click it or right-click and choose **Label entry** from the context menu. In the text box that appears, type a label manually. Alternatively, if the AI Innovations feature is enabled, use Gemini AI to generate a summary automatically.



  - **Connect two items:** If you notice two trace entries are relevant, for example a long network request linked to a specific component entry from the Main track, you can use the **Link entries** feature to link them together. To connect two relevant entries with an arrow, right-click the item and choose the **Link entries** option, then click the second item.

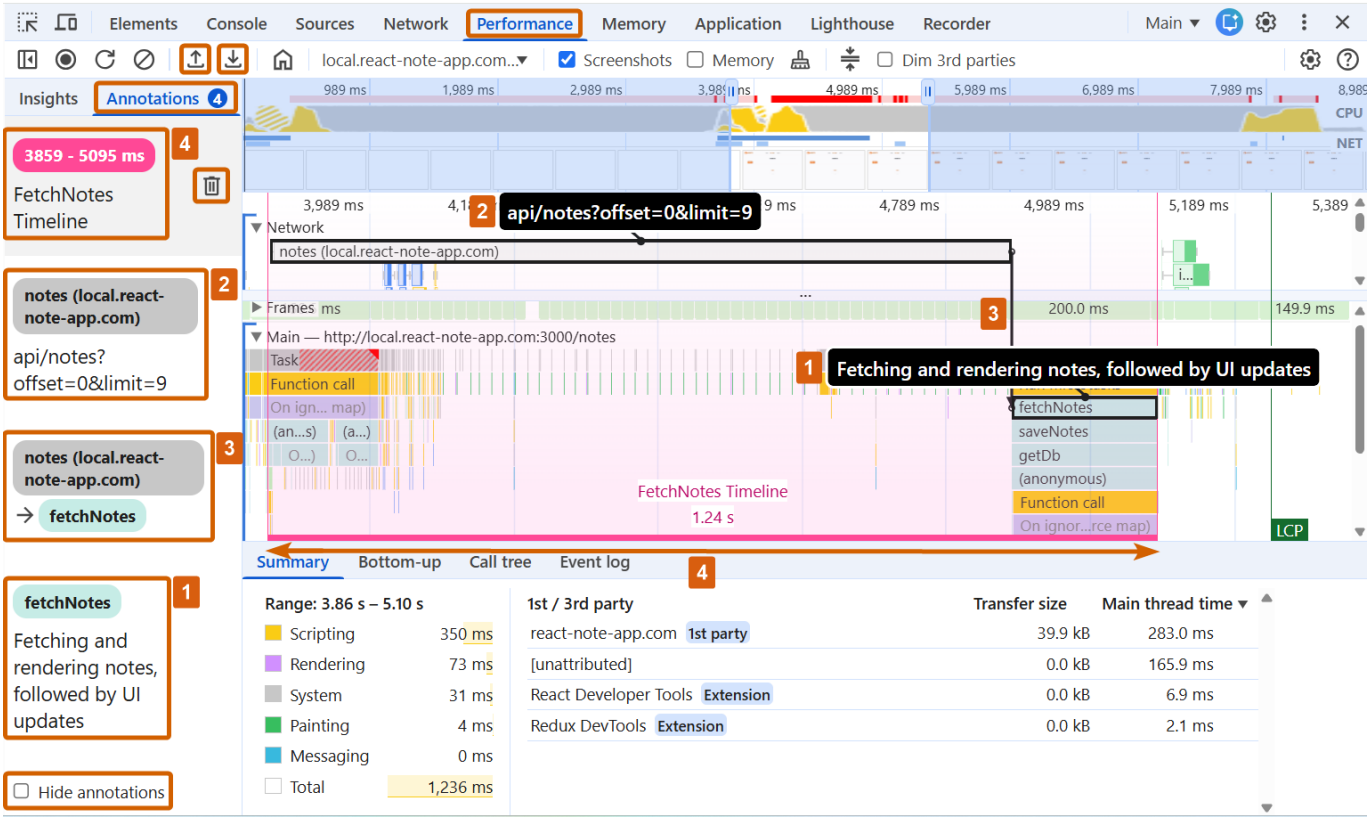


- **Label a time range:** To label an arbitrary time range, **shift-drag in flame chart** (Main track) from the start of a time range to its end, then type a label.



After exploring and adding annotations and links, you can:

- View created annotations in the **Annotations** tab based on your current debugging workflow.
- Hide created annotations using the **Hide annotations** checkbox.
- Remove an annotation by hovering over it in the **Annotations** tab and clicking the **Remove** icon.
- Export current trace using **Save trace... action (⭳)** to share with your team.
- Import performance traces provided by others using **Load trace... action (⭱)** to debug highlighted issues.



Performance Panel: Primary Sections

After a recording is complete, the panel displays a performance trace: a detailed timeline divided into different sections, each designed to help you analyze a specific aspect of runtime behavior.
Below are the primary sections essential for debugging performance bottlenecks. All the described information is shown either for the entire recording or for the selected timeline range.

> **Note:** Please refer to the screenshots in the Key Features and Workflow section above to see how the concepts described in this section are visualized in the DevTools UI.

Main (Main Thread) Track

- Shows your website's core runtime activity, like JavaScript execution, rendering and painting in a detailed **flame chart**.
  - **X-axis:** Represents the timeline of the recorded performance session. Wider bars indicate events that took longer to execute.
  - **Y-axis:** Represents the call stack. Events stacked vertically show a parent-child relationship, where upper events initiated lower events.
- Displays all work executed on the browser's main thread:
  - Function calls (with nesting)
  - Style recalculations
  - Layout
  - Paint

    - Garbage collection
- The *Long tasks** on the main thread (taking over `50 ms`) are flagged with a red triangle in the top-right corner.
- Useful for inspecting which events took the most time and understanding their call stacks.
  This is the primary and **the most important section for JavaScript profiling**, essential for identifying performance bottlenecks such as long execution tasks, slowdowns caused by heavy scripts, root causes of slow interactions (e.g., delays after user input, click or similar events), expensive timers/intervals, and much more.

Frames Track

- Shows how long each frame took to render, displayed in milliseconds.
- The actual important metric **(FPS - frames per second)*** is not shown directly in the Performance panel, but Chrome calculates it under the hood and visualizes frame quality using colors:
    - **Idle frame (white):** Represents time when the browser isn't actively rendering your page's content.
    - **Frame (green):** Rendered quickly enough for a smooth experience (normally `16.7 ms` or less).
    - **Slow frame (yellow):** May cause jank.
    - **Dropped frame (red):** Chrome could not render the frame in time.
- Whether FPS causes a janky experience depends on what you are testing: from the chart alone it is not always clear.
  For example, if you are testing heavy animations and the FPS does not reach **60 FPS** (the goal frame rate for the web), the browser is not rendering frames at the ideal rate and you will most likely experience jank.
- If you want to see the real-time FPS metric, you can use a specific tool available in Chrome DevTools: **Show Frames per Second meter**.
  To activate it, open the Performance panel and type **Show Frames per Second meter** in the **Command Menu**. The tool will show the FPS value in real time as you interact with the page.
- Useful for **spotting rendering hiccups such as janky animations and rendering delays**.

Screenshots Track

- Shows small thumbnail captures of the page.
- You can click a screenshot to jump to that moment in the timeline.
- Before starting the recording, make sure the **Screenshots** option in the Performance toolbar is enabled. Otherwise thumbnails will not appear.
- Useful for **visually connecting performance events with specific UI states**.

Network Track

- Shows the activity of network requests.
- Visualizes the start time, duration and execution order.
- **Hovering** over a request reveals essential details such as the URL, priority and a breakdown of **lifecycle phases** (e.g., **Content downloading**, **Waiting on Main thread** etc.) with their durations.

- **Clicking** on a request highlights its **initiators** using arrows, helping you trace the dependency tree of network requests.
- Useful for **detecting if delays are caused by waiting for API responses or other network resources**.

GPU Track

- Shows graphics-related work handled by the GPU process:
  - **Compositing layers**
  - **Rasterizing (drawing) layers***
    Note: There is a separate **Thread Pool** track in the Performance panel which shows rasterization work.
  - **Drawing compositor-driven animations** (e.g., `opacity` or `transform` animations that do not require Layout or Paint)
- All modern engines take advantage of GPUs for rendering, especially for compositing and rasterizing graphics-intensive parts, to achieve high frame rates and offload work from the CPU.
  For example, in Chrome, the concurrent raster threads typically run on the CPU to render pixels and then upload them to the GPU memory.
- Useful for **diagnosing rendering and animation performance issues**, especially jank caused by layer compositing or GPU workload.

Timings

- Displays markers for key browser or developer-defined events:
  - **DOMContentLoaded** (DCL)
  - **First Contentful Paint** (FCP)
  - **Largest Contentful Paint** (LCP)
- Developers can use the `performance.mark()` method, which is part of the `User Timing API`, to add developer-defined custom markers:

```
performance.mark("form-sent");
```

- Useful for measuring how long it took to reach specific marked events, and for **visually connecting them with other activities in the timeline**.

Layout Shifts Track

- Shows layout shifts.
- Shifts are shown as purple diamonds and are grouped in clusters (purple lines) based on their proximity on the timeline.
- Hover over the corresponding diamond to see which element caused the shift or click to view the summary.
- Useful when **identifying the root cause of the poor Cumulative Layout Shift (CLS) metric**.

Interactions Track

- Shows user interactions (e.g., link click, input).
- Marks the interactions over `200 ms` with a red triangle in the top right corner.
- Hover over the interaction to see a tooltip with input delay, processing time and presentation delay.
- Useful when **identifying the root cause of the poor Interaction to Next Paint (INP) metric**.

Animations

- Shows CSS animations.
- Animations are named by corresponding CSS properties or elements if any, for example, `transform` or `my-element`.
- When hovering over each animation item, the specific element will be highlighted in the UI, and when clicking on each animation item, its summary will be shown in the **Details View (Summary)** panel at the bottom.
- Non-composited animations are marked with red triangles in the top right corner. Non-composited animations are the ones that trigger one of the steps in the rendering pipeline: Style, Layout or Paint. They perform worse because they force the browser to do more work. These animations can also increase the Cumulative Layout Shift (CLS) of your page since they result in actual movement of elements that the CLS algorithm measures, which may cause cascading shifts to other elements. That's why they are marked specifically on the Performance panel and are not advised as good practice.
- Useful when **identifying long, broken and non-composited animations**.

Details View (Summary / Bottom-up / Call Tree / Event Log)

- Shows the details view of activities (e.g., `Layout`, `Layerize`, `Paint`, `mouse-up`, `click`, etc.).
- This panel can be found below the timeline when clicking an event or expanding the corresponding section. For example, in order to review activities that happened on the Main thread, you need to expand the Main track.
- Each view in the **Details View** gives you a different perspective on the activities:
    - **Summary:** Breakdown of time spent on different activities (**Scripting**, **System**, **Rendering**, **Painting**, **Loading**, **Messaging**) for the expanded section. Also provides summaries for specific event markers (DCL, FCP, LCP) or events (network request, animation) when clicked on.
    - **Bottom-up:** View the activities where the most time was directly spent. This includes all activities including root activities and framework initiated activities. The activities are ordered by **Self-time** which is the individual time spent on specific activity excluding children.
    - **Call Tree:** View the *root activities** that caused the most work. The activities are ordered by **Total time**, which is the aggregated time spent on the activity including children.
    - **Event Log:** View the activities in the order in which they occurred during the recording or the selected timeline range.
- If you click on any event like animation, long task, or network event, the Details View also updates and will display information relevant to the selected event.
- You can also use filtering and grouping options in the Details View to view only necessary events. For example, grouping by **Third Parties** allows you to calculate total time spent on Third-party libraries

loading and consider optimizations.
- Useful for **quickly identifying which part of rendering or which resource specifically caused performance issues**. This is a primary tool for reviewing how much time each event took and comparing after optimizations are done.

Performance Panel: Custom Tracks

The Performance panel empowers you to bring your own essential project performance data directly into the timeline to help you debug your complex applications more easily. Whether you're a framework author or a developer building complex applications, you can create your custom tracks, performance measures, and markers in the Performance Panel through the **User Timing API** and the **Chrome DevTools Extensibility API**.

Framework Powered Custom Tracks

Many frameworks (such as **Next.js**, **Angular**) integrate with the **Chrome DevTools Extensibility API** to present framework-specific data and insights directly in the **DevTools Performance panel of Chromium-based browsers** (e.g., Google Chrome, Microsoft Edge). These tracks usually have framework-specific names and are marked as **Custom**.
Some frameworks like **Next.js** enable Performance custom tracks by default, while for others like **Angular**, you need to enable them.
You can learn more about React and Angular specific custom performance tracks in the upcoming sections: React Custom Performance Tracks and Angular Custom Performance Track.

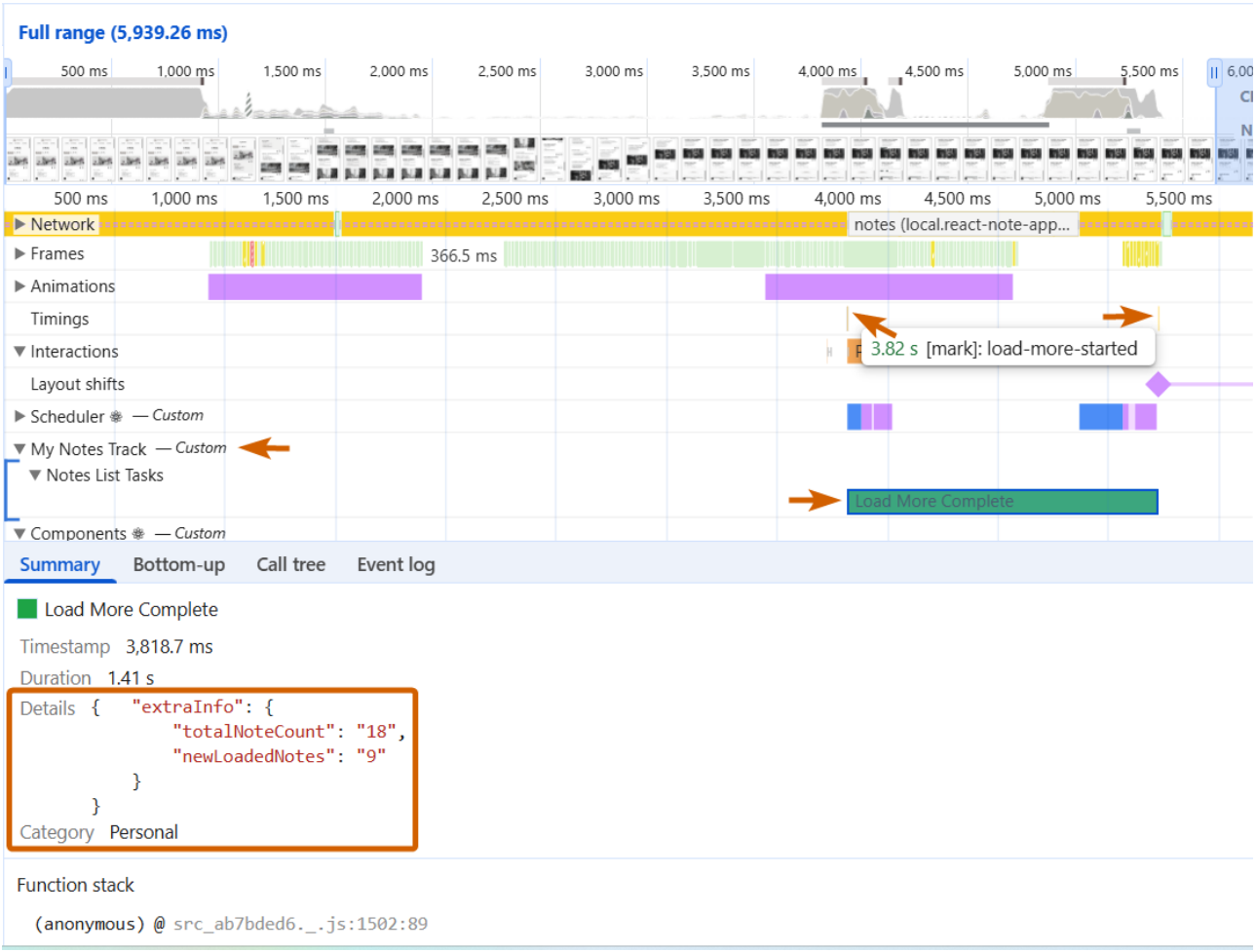
How to Add a Custom Track

Custom tracks can be initialized by developers to carry essential project data for further debugging. However, this requires some effort on the developer side:

- Use the `performance.mark()` method from the **User Timing API** to visually highlight specific points of interest in the timeline with custom markers that span across all tracks.
- Use the `performance.measure()` method from the **User Timing API** to create a named `PerformanceMeasure` object representing a time measurement between two marks in the browser's performance timeline.
- For browsers that support the **Extensibility API**, you can use the `detail` property in `performance.measure()` to provide more details in a `devtools` object that will be used to display **custom tracks** and **custom track groups** with the specified details in performance profiles:
    - Define custom track group names and custom track names.
    - Add extra metadata that will be displayed in the Summary tab.
    - Add color, extra properties (e.g., selected category) and tooltip.

Example:

```
// Add `load-more-started` marker
performance.mark('load-more-started', {
  detail: 'Load more: loading 9 notes.',
});

// Do load more data fetch work

// When fetch is finished, add performance measure
// between `load-more-started` and `load-more-ended` markers
// with a title `Load More Complete` on custom track `Notes List Tasks`
// in a custom track group: `My Notes Track`
performance.measure('Load More Complete', {
  start: 'load-more-started',
  end: 'load-more-ended',
  detail: {
    // This data appears in the Summary tab
    extraInfo: {
      totalNoteCount: '18',
      newLoadedNotes: '9'
    },
    devtools: {
      dataType: 'track-entry',
      track: 'Notes List Tasks',
      trackGroup: 'My Notes Track',
      color: 'tertiary-dark',
      properties: [
        ['Category', 'Personal'],
      ],
      tooltipText: 'New Notes Loaded Successfully',
    },
  },
});

// Add `load-more-ended` marker
performance.mark('load-more-ended', {
  detail: 'Load more: loaded 9 notes.',
});
```



Custom Tracks: Use Cases

- Debugging framework-specific entries contributed by the framework's runtime.
- Introducing project-specific custom tracks for your complex application to include specific performance data (e.g., the state of your application or information about a specific user type) for more effective debugging.

Performance Panel: Memory Debugging

The Performance panel also allows you to debug memory allocation during the process of recording a performance profile.

To enable memory debugging, click the **Memory** checkbox in the **Capture settings** toolbar at the top. A new **Memory chart** will appear in the **Performance** panel section at the bottom, above the **Summary** tab.

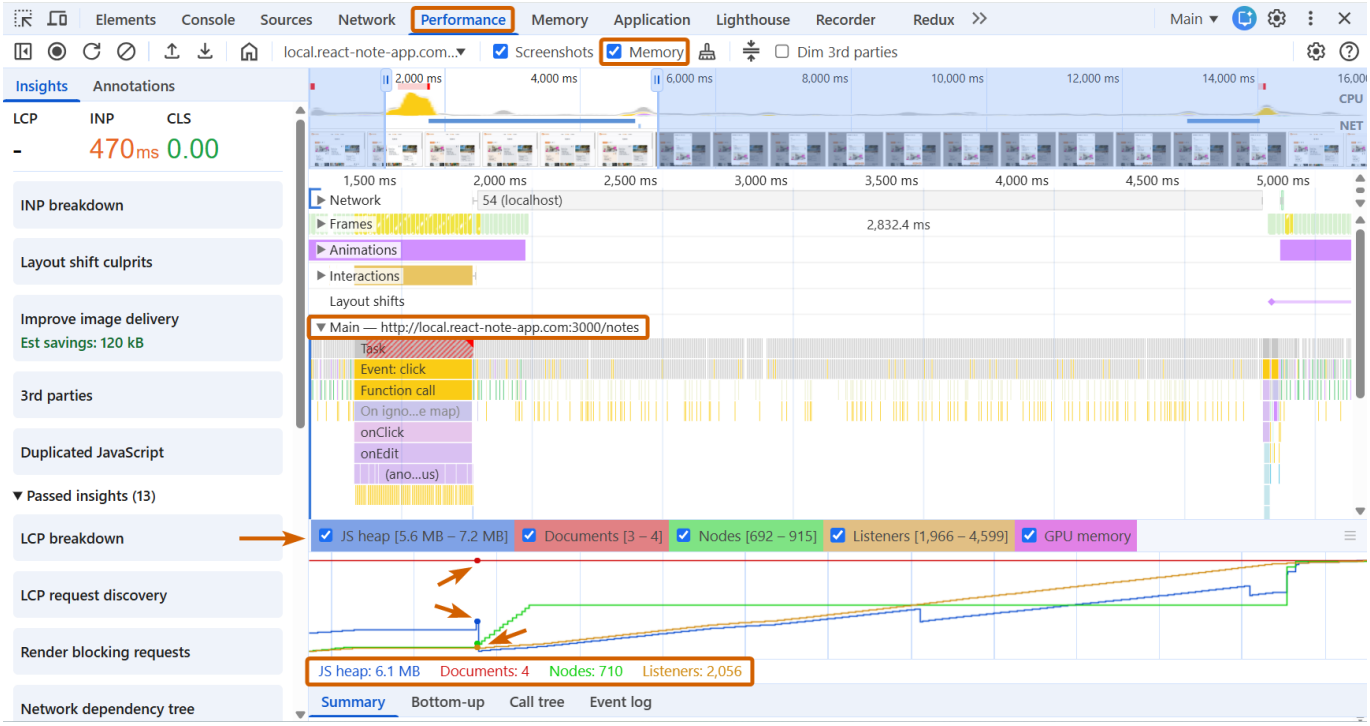
In order to view memory usage associated with the specific track, you need to expand that track. The best way is to start by expanding **Main Track (flame chart)** and view the memory usage at a glance across the recording timeline.

The **Memory chart** allows you to view the following information across the performance trace timeline:

- ***JS Heap**** **allocation size**
- **Page Documents count**

- **DOM Nodes count**
- **Event listeners count**
- **GPU memory size**

Each metric will be shown in its own **line chart**, colored with its specific color. You can hover over the lines, and view at the bottom the summary of memory usage. This is a useful feature because it helps you quickly identify relatively high memory usage and *memory leaks** during specific, even short time intervals. Most importantly, you can correlate this data with other tracks to identify what caused the leak.



While the Performance Panel remains the most important and useful panel for debugging the performance of your applications, there are some other useful tools that Chrome DevTools offers to accelerate your application memory debugging flow.

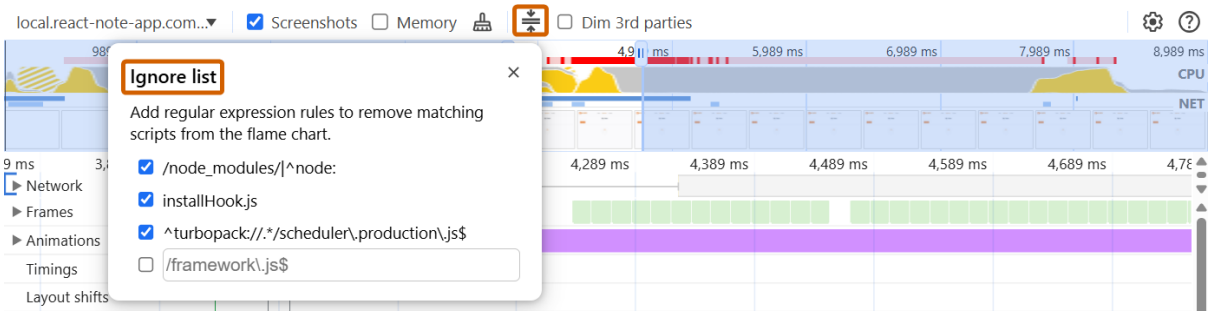
- **Performance Monitor** panel displays a timeline that graphs performance metrics in real-time (**CPU usage**, **JS heap size**, **DOM Nodes**, etc.). You can click a metric to show or hide it and observe how the graph changes as you interact with your application.
- **Memory** panel provides diagnostic tools that let you see the memory distribution of JavaScript objects, discover and isolate memory leaks, get a breakdown of memory allocation by function, and more. It allows you to record a profile based on different perspectives (e.g., **heap**, **detached elements**, **memory allocations**, etc.).

To open these panels, open DevTools and type the corresponding panel name from the **Command Menu**.

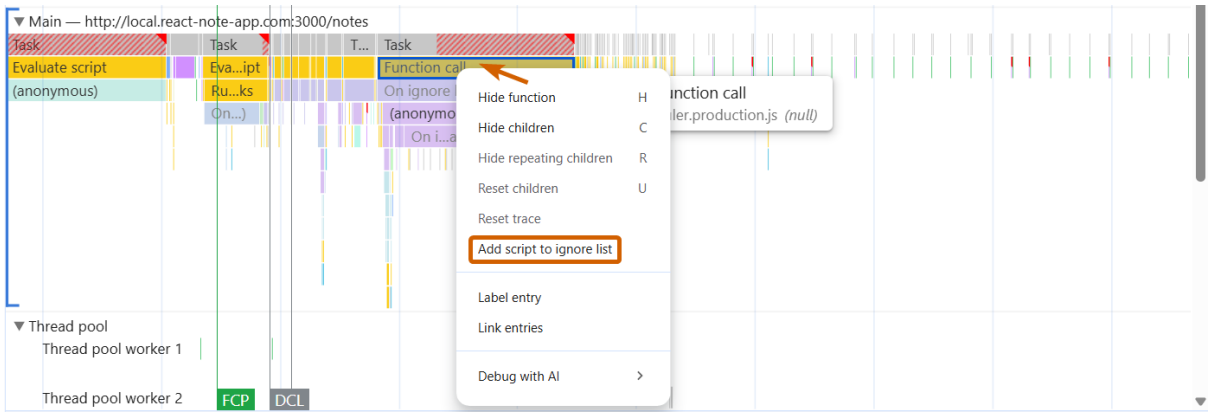
Performance Panel: Useful Debugging Tips

Performance traces can become large and complex, making them difficult to read and interpret. Below are several tips that help simplify the trace and make performance debugging more manageable:

- **Zoom into a specific area:** Use mouse zoom-in/zoom-out features to zoom into a specific area of the performance trace for better visibility.
- **Select timeline range:** Use timeline markers to choose a specific timeline range and review performance trace details within that range.
- **Enable screenshots capturing:** Always enable the **Screenshots** checkbox before recording a profile. If enabled, it generates screenshots alongside the timeline, helping you easily connect what happened in the UI during specific events to activities in different tracks.
- **Add unnecessary files to ignore list:**
    - Use the **Show Ignore List setting** icon on the top **Capture settings** toolbar to add some files and file patterns from the **Main track** to the Ignore List. This helps to make the performance trace more clear and understandable.

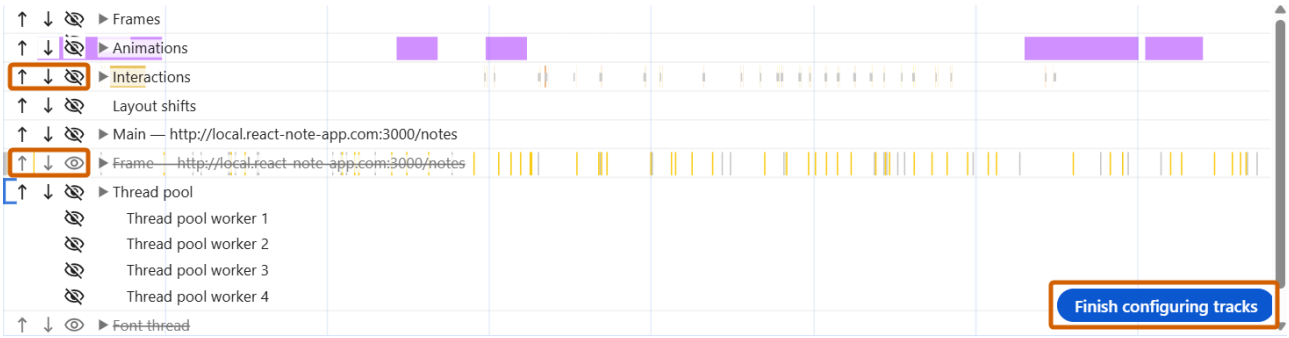


    - Alternatively, you can right-click on any script in **Main track** (flame chart), and choose **Add script to ignore list** or **Remove script from ignore list**.



For example, you may want to ignore framework-specific entries in order to concentrate on the root cause of performance issues.

- **Hide unnecessary tracks:** You can hide some tracks and leave only essential ones to simplify your debugging process. Right-click on any track and click on **Configure tracks**. In the opened dialog, choose which tracks to show and which ones to hide, then press **Finish configuring tracks** to save the results.



Performance Panel: Common Use Cases

- **Finding Performance Bottlenecks**: Identify the root cause of slow interactions (high INP) or poor page load speed (high LCP):
    - Check for any **render-blocking requests** in the **Performance panel > Insights pane**.
    - Analyze long tasks in the **Main track** and correlate them with entries in **Screenshots**, **Network**, **Interactions**, **Layout Shifts** and **Framework/developer-specific custom tracks** to identify the root cause.
- **Excluding Third-party Scripts and Comparing Results**: Exclude third-party and specific first-party scripts and check how it affects the overall performance of your application.
    - Filter **Details View > Bottom-up** tab by **Group by Third Parties** option to see which third party scripts take a long time.
    - Make a performance recording and consider it as your **baseline**.
    - Use the **Network request blocking** panel in DevTools to block third-party resources that take a long time, one by one.
    - After blocking each resource, record a new profile and use the **Show recent timeline sessions dropdown** in the **Performance** panel top **actions toolbar** to compare the new recording with your **baseline**. Check the **Details View > Summary** tab, **Core Web Vitals** and other metrics across both traces to identify performance gains.

  Consider removing or replacing some third-party scripts based on your performance findings.
- **Debugging Jank Rendering and Inefficient Animations**:
    - Check **Animations track** for animations that are marked with **red triangle**, which indicate non-composited/slow animations and can be potentially optimized.
    - Check **Main track** for **Recalculate style** and **Layout** sequential blocks which are marked with **red triangle**. These are cases of **forced reflow** and are potentially affecting your application performance.
    - Watch out for dropped frames (rendered in `> 16ms`) in the **Frames** track which likely indicate janky experiences.

  For example, an animation that moves an element using something other than `transform` (like `top` and `left`) is non-composited and likely to be slow.

To improve performance of non-composited animations, you can use `will-change: transform;` CSS property to force layer creation. Because layer creation can cause other performance issues, it is not recommended to use it early in your optimization process. Instead, use it only if you notice visual performance issues.

- **Identifying Unused or Inefficient Code**: Use the **Main track** combined with the DevTools **Coverage** panel in order to discover JavaScript that is running unnecessarily or taking too long to execute, which can be refactored or removed to speed up the main thread.
    - Start unused code recording in the DevTools **Coverage** panel by clicking on the **Instrument coverage** (●) icon.
    - Make a Performance panel recording and consider it as your **baseline**.
    - Identify scripts that take a long time in the **Main track**. Hover over the script to find the source file path.
    - Locate the source file of the script that was taking a long time in the **Sources** panel and check its unused code coverage (highlighted in red left border).
    - Remove unused code if it is not used anywhere, or if used, you can defer or load it only during necessary interactions.
    - Record a new performance profile and compare it with the **baseline** to notice improvements in metrics.

- **Annotating and Sharing Performance Traces**:
    - Record a trace and **annotate** it by marking specific areas with labels or linking two different entries.
    - **Export** the trace file to share with your team or attach to a bug ticket.

  This allows other developers to import the recording into their own DevTools for further analysis and troubleshooting.

- **Debugging Memory Leaks:**
    - Use the Performance panel **Memory** feature to debug **JS Heap allocation size**, **Page Documents count**, **DOM Nodes count**, **Event listeners count** and **GPU memory size** across the timeline of a selected recording.
    - Identify timeline ranges where memory constantly and abnormally grows, causing *memory leaks**. Correlate these time ranges with the events happening in the UI via the **Screenshots** track and other tracks (e.g., **Main track**) to identify the root cause of memory leaks.

  A standard accepted way of avoiding memory leaks is unsubscribing from observables (e.g., in **Angular RxJS**), clearing time intervals and any other resources that are not handled by default by the JavaScript engine garbage collector after components are destroyed.

> **Note: Network Request Blocking** and **Coverage** are separate panels in **Chrome DevTools**. After opening **DevTools**, you can use the Command Menu to quickly find and open these panels.

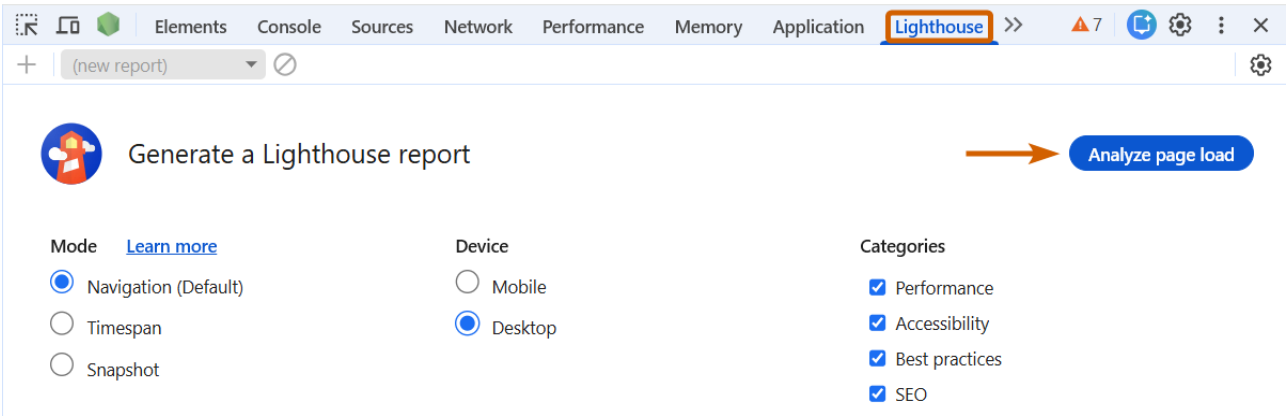
1.5 Lighthouse Panel

The **Lighthouse** panel provides an integrated performance auditing tool, based directly on Google's Lighthouse tool, to help you analyze and improve your page's **performance**, **accessibility** and **usability**.

Lighthouse Panel: How to Analyze

- Open DevTools and click the **Lighthouse** tab or select it from the **Command Menu**.
- Choose the criteria such as **Device** (Desktop or Mobile), necessary **Categories** and the **Mode** to be used for the analysis. The **Mode** provides the following options:
    - **Navigation (Default):** Click on the **Analyze page load** button, and the tool will reload your application and analyze its performance during page load.
    - **Timespan:** Click on the **Start timespan** button, perform some actions in your application, and then click **End timespan**. The tool will analyze the performance of your application during the specified timespan.
    - **Snapshot:** Click on **Analyze page state** and the tool will analyze the current state of your application.

    Obviously, the **Navigation (Default)** mode gives the most useful and richest results for your application analysis.

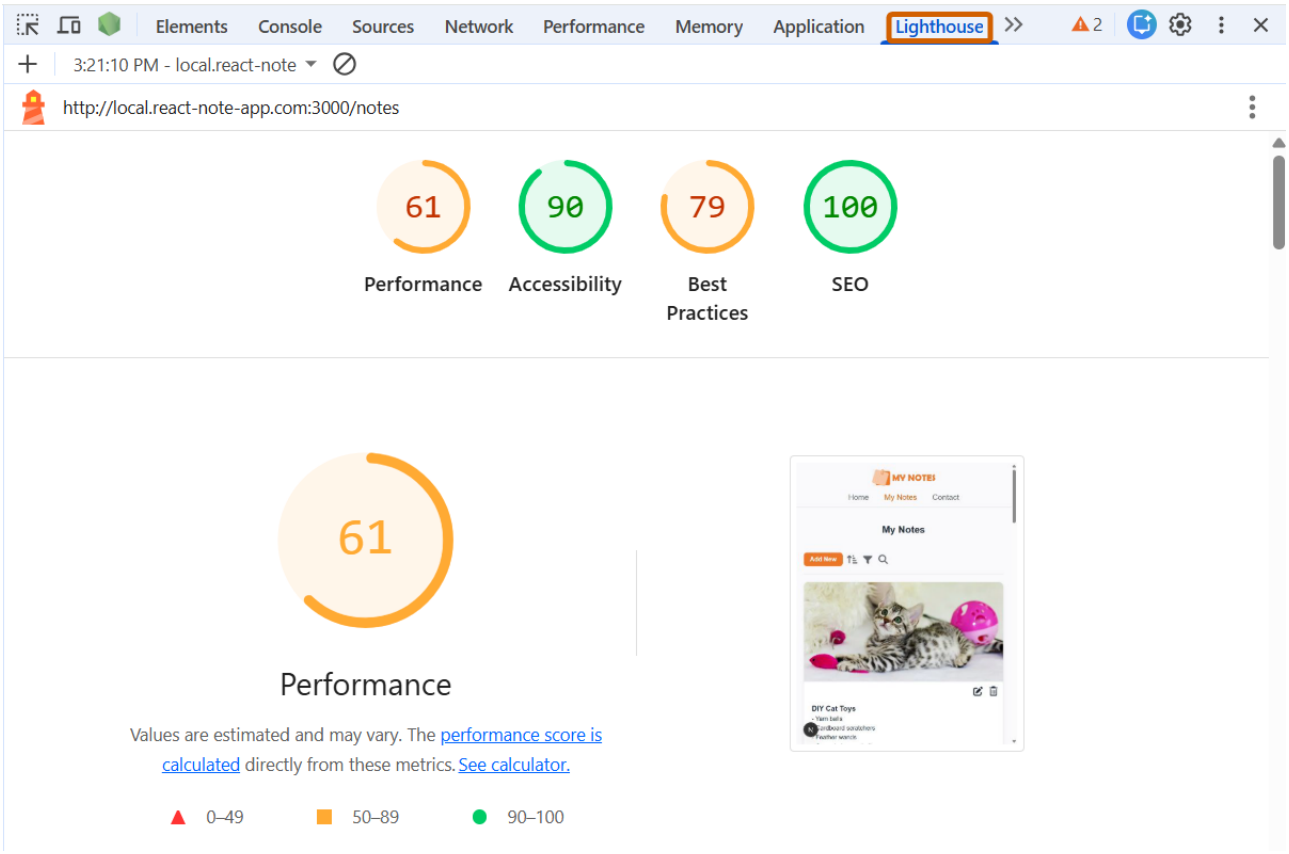


- View the report and optimization suggestions.

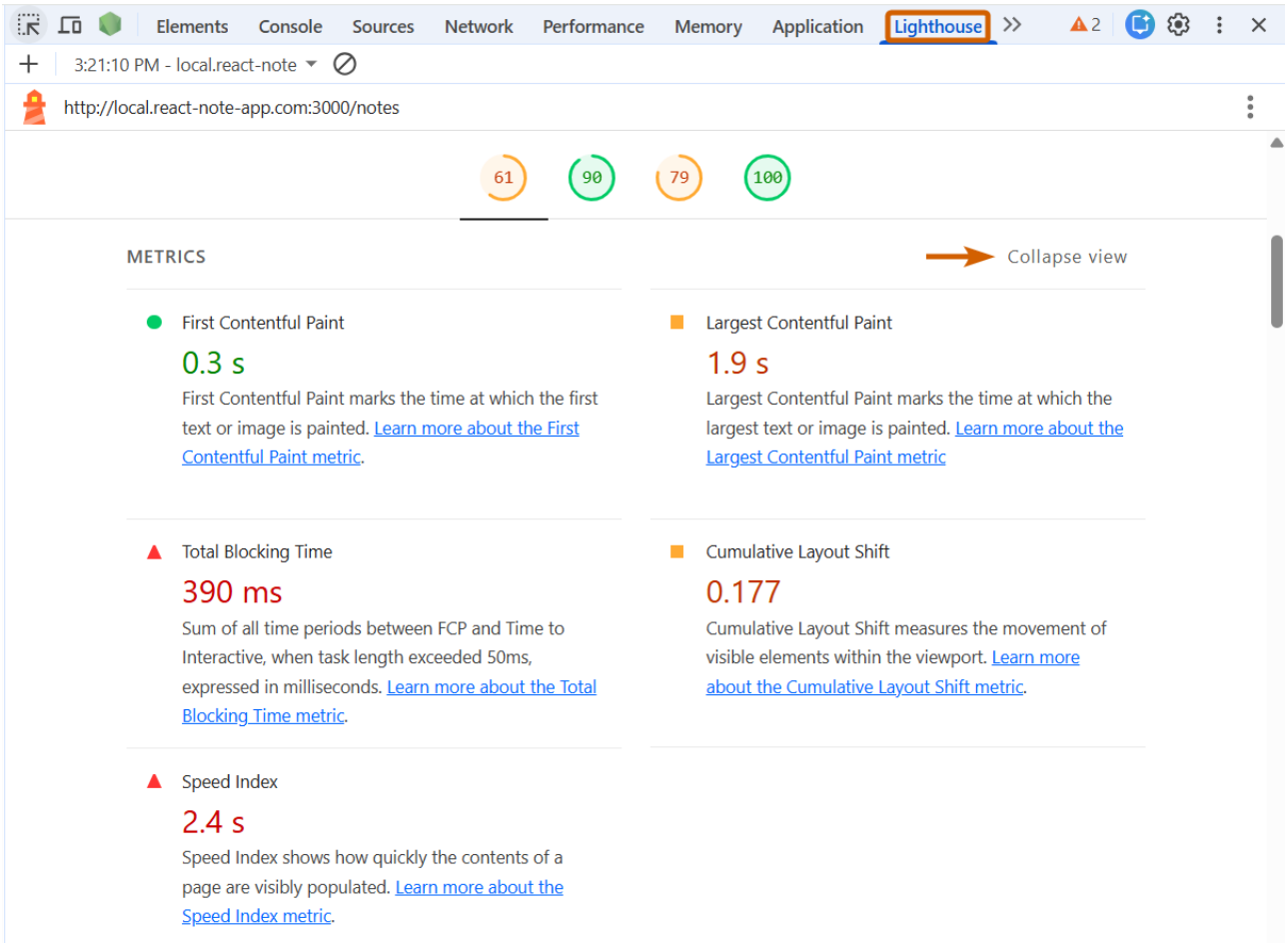
Lighthouse Panel: Key Features

After the report is ready, you can view the **important information and optimization suggestions** it provides to help you optimize your application:

- **Performance Summary**: Provides a summary of four core performance categories: **Performance**, **Accessibility**, **Best Practices** and **SEO**.



- **Performance Metrics**: Shows performance metrics of your application (including three **Core Web Vitals** metrics also available in the **Performance** panel).

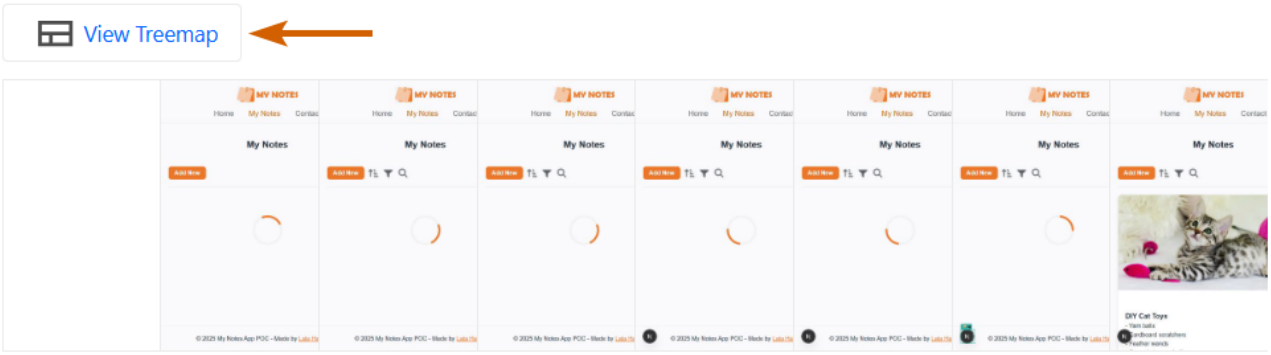


- **First Contentful Paint (FCP):** Marks the time at which the first text or image is painted.
- **Largest Contentful Paint (LCP):** Marks the time at which the largest text or image is painted.
- **Total Blocking Time (TBT):** Sum of all time periods between **FCP** and **Time to Interactive**, when task length exceeded `50 ms`, expressed in milliseconds.
  The **TTI (Time to Interactive)** metric measures the time from when the page starts loading to when its main sub-resources have loaded and it is capable of reliably responding to user input quickly.
- **Cumulative Layout Shift (CLS):** Measures how much the page content unexpectedly shifts during loading.
- **Speed Index (SI):** Measures how quickly content is visually displayed during page load.
  Your Speed Index score is a comparison of your page's speed index and the speed indexes of real websites, based on data from the HTTP Archive.

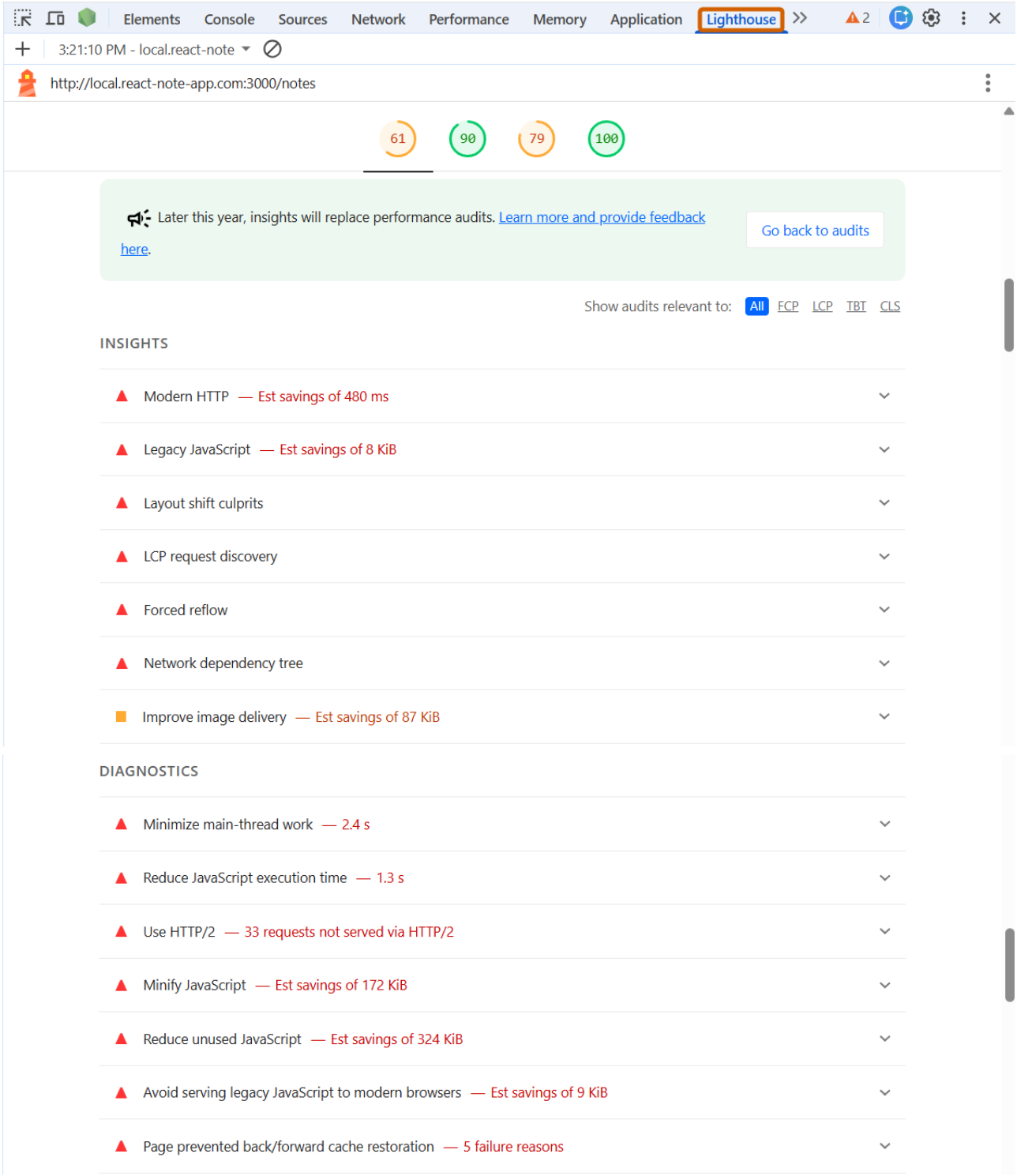
You can click **Expand view** to see a detailed explanation of each metric and understand how it is evaluated. Clicking on **Collapse view** will close the details accordingly.

For more explanation of **Core Web Vitals** metrics, check Performance Panel: Key Features and Workflow section.

- **View Treemap:** Allows you to dive deeper into the JavaScript bundles on a page by clicking **View treemap**. This opens the page's bundles in the Lighthouse Treemap where you can view **Unused bytes** and **Duplicate modules.** This is a very useful and often overlooked feature.



- **Actionable Fixes**: The **Insights** and **Diagnostics** sections provide details about specific issues affecting your score and offer clear, actionable recommendations on how to resolve them.
  The **Insights** section is new and shows the same performance insights available in the **Performance** panel. The goal was to offer the same performance advice across all Google performance tools: Chrome DevTools, Lighthouse and PageSpeed Insights.
  This means some old audit metrics were removed and replaced with new insights-based audits shown under the **Insights** heading, while the unchanged audits continue to appear under the **Diagnostics** heading.



- **Consistent Auditing**: The analysis is powered by the same engine as the Lighthouse tool (separate Chrome extension by Google), ensuring you get reliable and standard-compliant results.

Lighthouse Panel: Common Use Cases

- **Getting an Actionable Performance Report**: Run a performance audit to get a score and, more importantly, a list of specific **Insights** and **Diagnostics** that show you exactly how to speed up your page.
- **Auditing for Accessibility (a11y)**: Check for common accessibility issues like color contrast problems, missing ARIA attributes and improper heading structures to make your application more usable for

everyone.

- **Auditing for SEO Best Practices**: Verify that your page is optimized for search engine crawlers by checking for things like a valid `robots.txt`, a meta description and descriptive link text.
- **Checking for Web Best Practices**: Get a report on general web hygiene, ensuring your page uses HTTPS, is free from browser errors and follows other modern web standards.
- **Diagnose Unused Code**: Use the **View treemap** option to diagnose **unused bytes** and **duplicate modules** in JavaScript bundles on a page and get rid of duplicate and unused code.

Note:

- To get correct statistics, it is better to generate a Lighthouse report for your **local production build**, because development builds normally contain a lot of heavy code (e.g., Next.js dev build) to assist your local development process and may not give you an accurate report.
- While the **Lighthouse panel** in Chrome DevTools is really powerful, combining audits from the **Lighthouse tool** and **PageSpeed Insights**, you can also use these tools separately for your **hosted web application**.
  For that, you need to activate the **Lighthouse browser extension** available for Chrome and Firefox browsers, which will allow you to run both Lighthouse and PageSpeed Insights reports. **PageSpeed Insights** can also be accessed without the Lighthouse extension by visiting pagespeed.web.dev.
  The main difference is that the extension and the PageSpeed Insights platform cannot run on your local application. For that, you need to rely on the Lighthouse panel in DevTools.

1.6 Network Panel

The **Network** panel is an essential tool for debugging network activity. It lets you inspect all resources loaded by a webpage, analyze API calls and diagnose performance issues by showing you exactly what's being requested and received.

Network Panel: How to Open

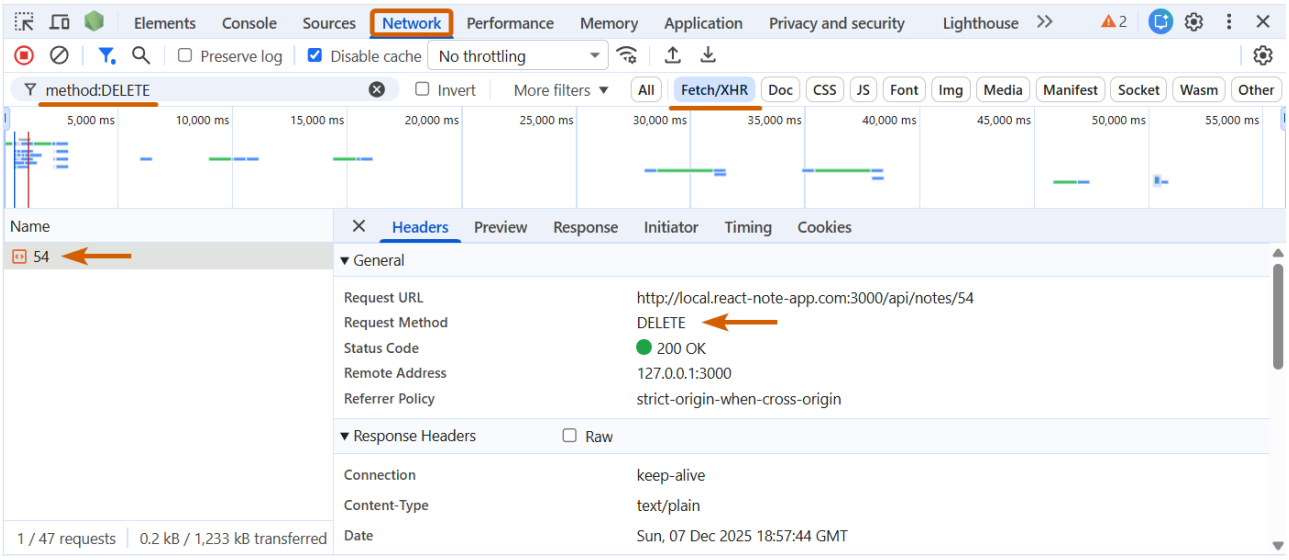
To open the **Network** panel, open DevTools and click the **Network** tab, or select it from the **Command Menu**.

Network Panel: Filtering Requests

The Filter bar allows you to filter requests by specific criteria:

- **Filter by Resource Type**: Click on filter buttons such as **Fetch/XHR** to see only API requests, or **Img** to see only images. This is very useful when you need to isolate specific resource-related issues (e.g., API failures).

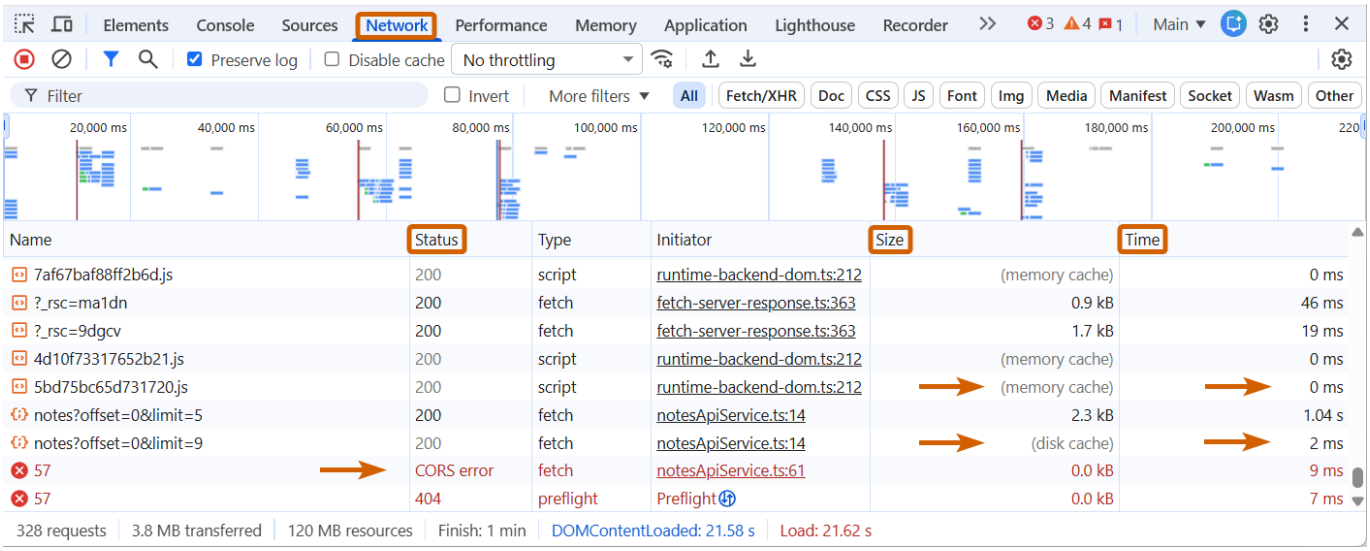
- **Filter by Text**: You can type text into the filter box to find requests with matching names, types or paths (e.g., `api/users`).
  For example, start typing filter keywords like `method:` and it will autocomplete all available options for you. Choose from the available list and filter your network requests based on a specific option (e.g., `DELETE`).
  This is particularly useful for **REST APIs**, when requests for the same entity look identical on the Network panel. For instance, there can be `DELETE`, `PUT` and `GET` requests with the same `api/notes/54` URL, and all of these requests will show only `54` in the Network panel, making them difficult to differentiate.



Network Panel: Requests Table

The main **Requests table** displays all logged network requests and provides useful at-a-glance information to quickly spot issues. Below are some useful default columns to help you to start debugging:

- **Size:** Shows the total size of the response (headers plus body) as delivered by the server.
    - If it shows `(disk cache)` or `(memory cache)`, it means the resource was not re-downloaded, confirming that your caching strategy is working.
    - If it shows a specific number, it means the resource was downloaded and not cached. In this case, it helps to notice at a high level which resources took the most transfer size for further optimizations.
- **Time:** Shows the total duration from the start of the request until the final byte of the response is received.
    - If the resource was served from local cache, it will show minimal time (e.g., `0 ms` or `1 ms`) for static resources.
    - If the resource was downloaded and not cached, it can help you to notice at a high level which resources took the most time to download for further optimizations.
- **Status:** Shows HTTP status code (e.g., `200` or `401`) of the request along with additional information (e.g., CORS error).
  Since failed requests are also highlighted in red, this column can help you quickly identify API failures and their root causes.

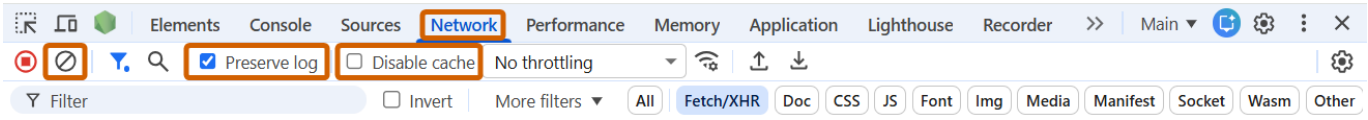


Apart from the default columns displayed in the **Requests table**, you can also right-click on the table header and choose to show/hide columns from a list of available options.

There are also the following useful actions provided in the **Requests table** top toolbar:

- **Preserve Log:** If checked, DevTools saves all network requests across page loads until you disable Preserve log.

- **Clear network log (⃠):** Clears the Requests table data, allowing you to log a fresh copy of network requests.
- **Disable cache:** If checked, DevTools disables the browser cache. This can help you emulate how a first-time user experiences your web application. It is also useful during local development to prevent requests from being served from the browser cache on page reloads.

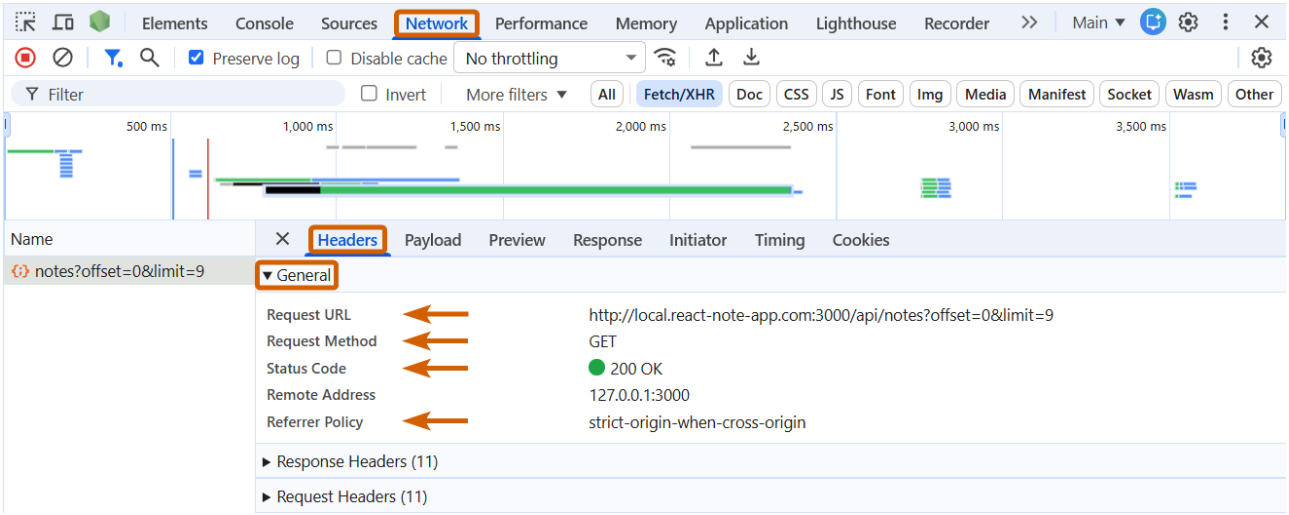


Network Panel: Inspecting Headers and Payloads

You can click on any request in the **Requests table** and explore different tabs for in-depth analysis of HTTP header data:

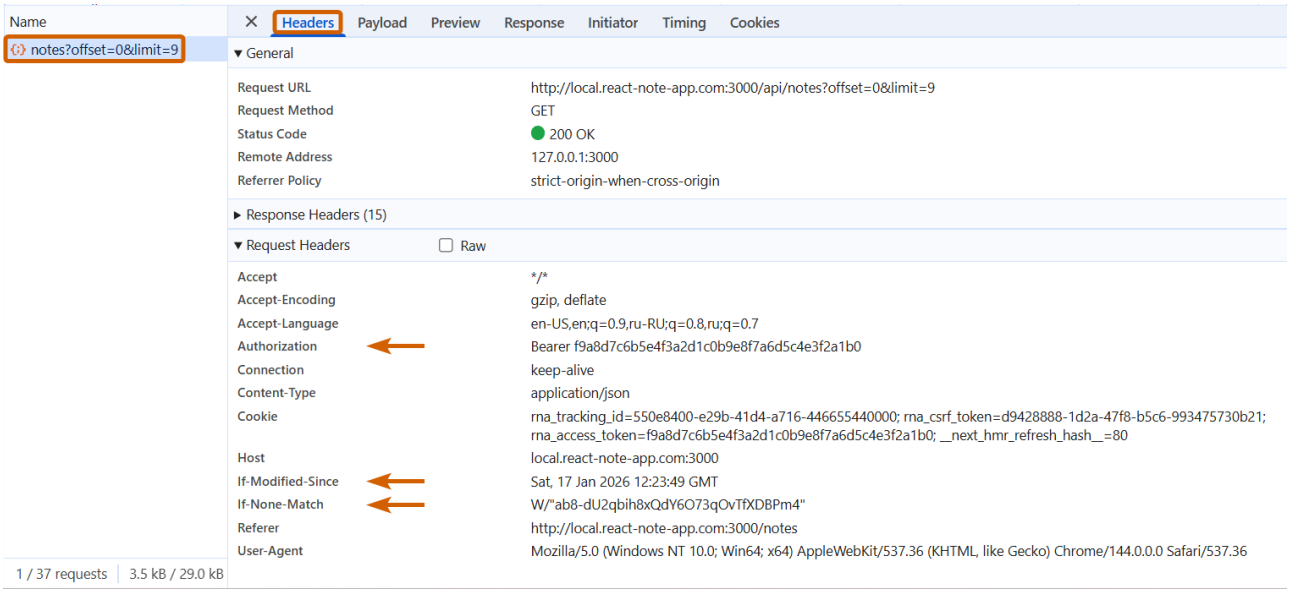
Headers: This tab shows general information about the request as well as a detailed breakdown of request and response headers. It is divided into the following sections:

- **General**: View the request URL, request method (e.g., `GET`, `POST`, `DELETE`), status code and referrer policy.
  Check the Common HTTP Status Codes: Cheat Sheet section below to learn more about common HTTP status codes.



- **Request Headers**: Expand the **Request Headers** section. Here you can verify critical information, such as whether an `Authorization: Bearer <token>` header, cookies, or the `If-None-Match` and `If-Modified-Since` cache synchronization headers are being correctly sent with your API calls.
  - **`Authorization: Bearer <token>`** is an HTTP request header used for API authentication, where the `<token>` is a security key (often a JWT) granting access to protected resources. Check the Debugging Access Tokens section below to learn more about how to debug authorization tokens.
  - **`If-None-Match`** and **`If-Modified-Since`** headers are used by the client to include the previously stored `ETag` or `Last-Modified` header values and send them as request headers to

the back-end for cache synchronization.
Check the Debugging Caching Headers section below to learn more about `If-None-Match` and `If-Modified-Since` headers.



- **Response Headers**: Expand the **Response Headers** section. Here you can check for caching-related headers like `ETag`, `Cache-Control` and `Last-Modified`, view cross-origin resource sharing (CORS) headers like `Access-Control-Allow-Origin` and `Access-Control-Allow-Credentials` or CSP policy header `Content-Security-Policy`.

    - **`ETag`**, **`Last-Modified`** and **`Cache-Control`** are the main headers used for controlling caching policy of your application's static assets.
      Check the Debugging Caching Headers section below to learn more about `If-None-Match` and `ETag` headers.

    - **`Access-Control-Allow-Credentials`** tells browsers whether the server allows credentials (e.g., cookies, authentication headers) to be included in cross-origin HTTP requests.

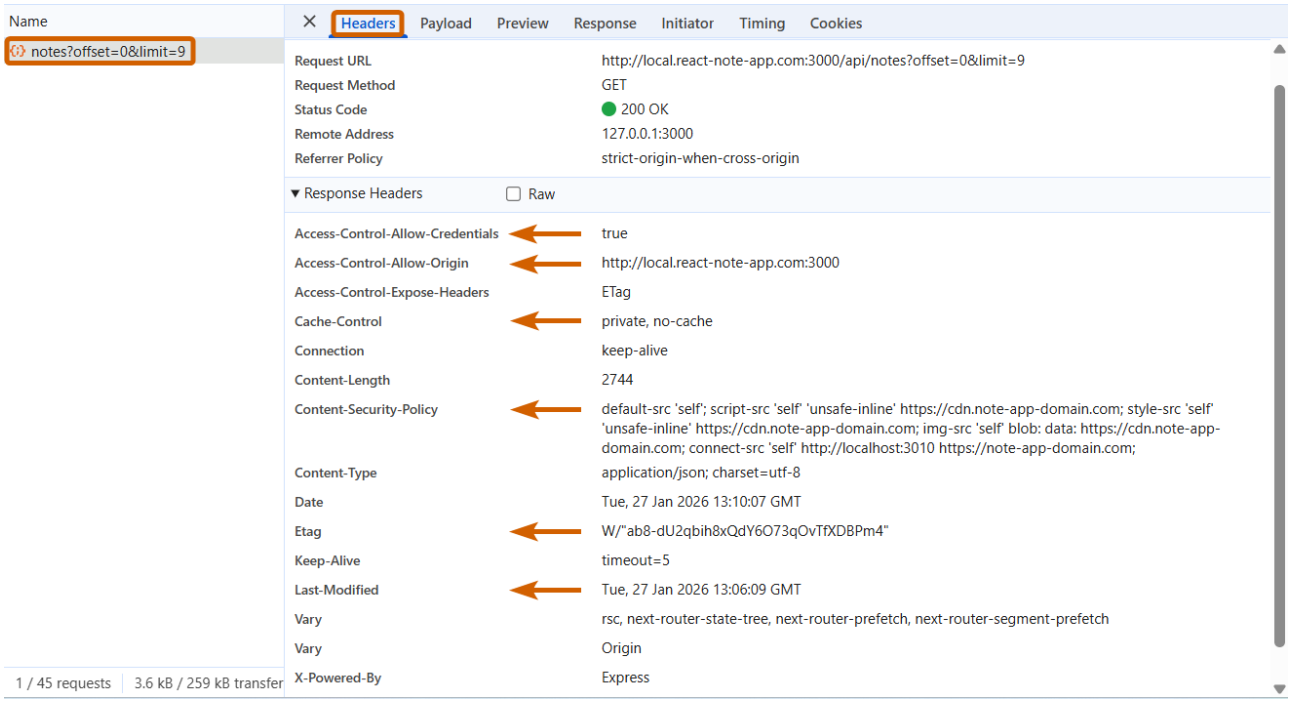
    - **`Access-Control-Allow-Origin`** indicates whether a resource can be accessed by web pages from a different origin.
      Check the Debugging CORS Errors section below to learn more about CORS policy and `Access-Control-Allow-Origin` header.

    - **`Content-Security-Policy`** allows you to restrict which resources (e.g., JavaScript, CSS, images) can be loaded for a given page (HTML document). It is an important security header which helps to protect against XSS attacks, where attackers try to inject malicious code into your page or load malicious third-party resources.

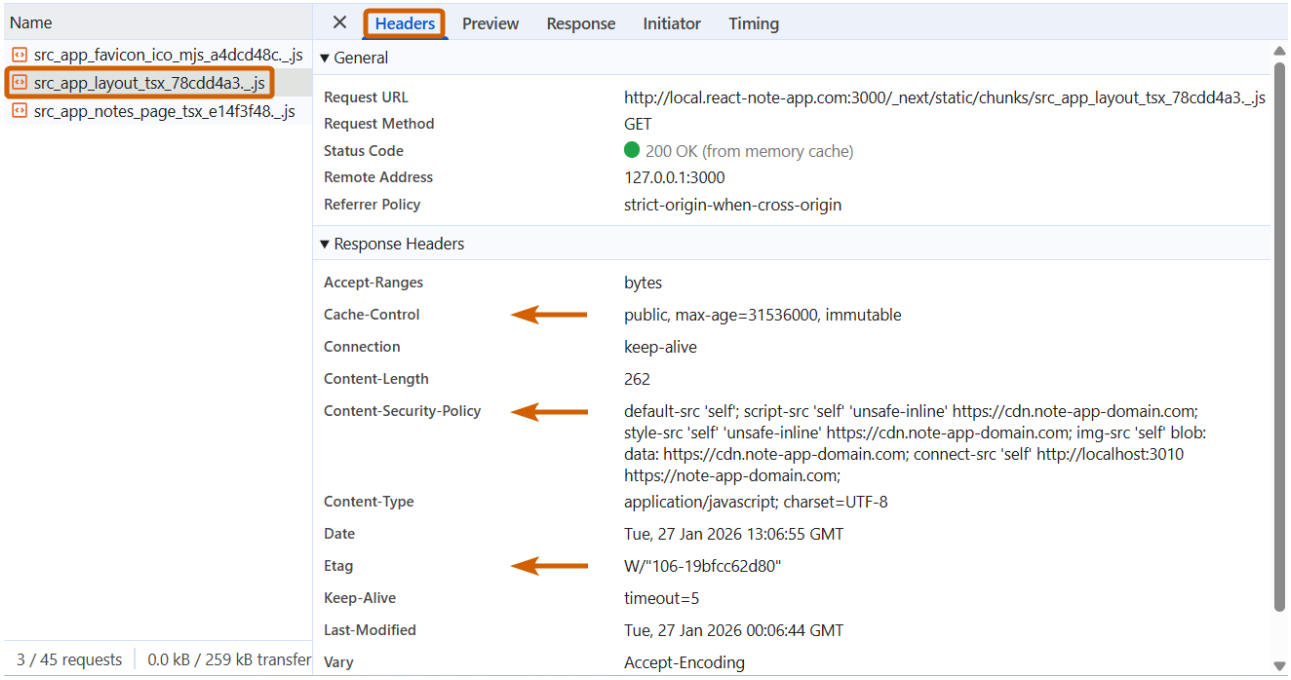
      Below is an example of a simple CSP policy for static resources (JavaScript, CSS and images) and API that allows only resources from the same origin (`self`), specific CDN resources, inline scripts as well as API requests from specified domains. You can consider making the inline script rule (`unsafe-inline`) stricter based on the framework of your application.

```
default-src 'self'; script-src 'self' 'unsafe-inline' https://cdn.note-app-domain.com; style-src 'self' 'unsafe-inline' https://cdn.note-app-domain.com; img-src 'self' blob: data: https://cdn.note-app-domain.com; connect-src 'self' http://localhost:3010 https://note-app-domain.com;
```

Example of API Request Response Headers:



Example of Static Resource (JS File) Response Headers:



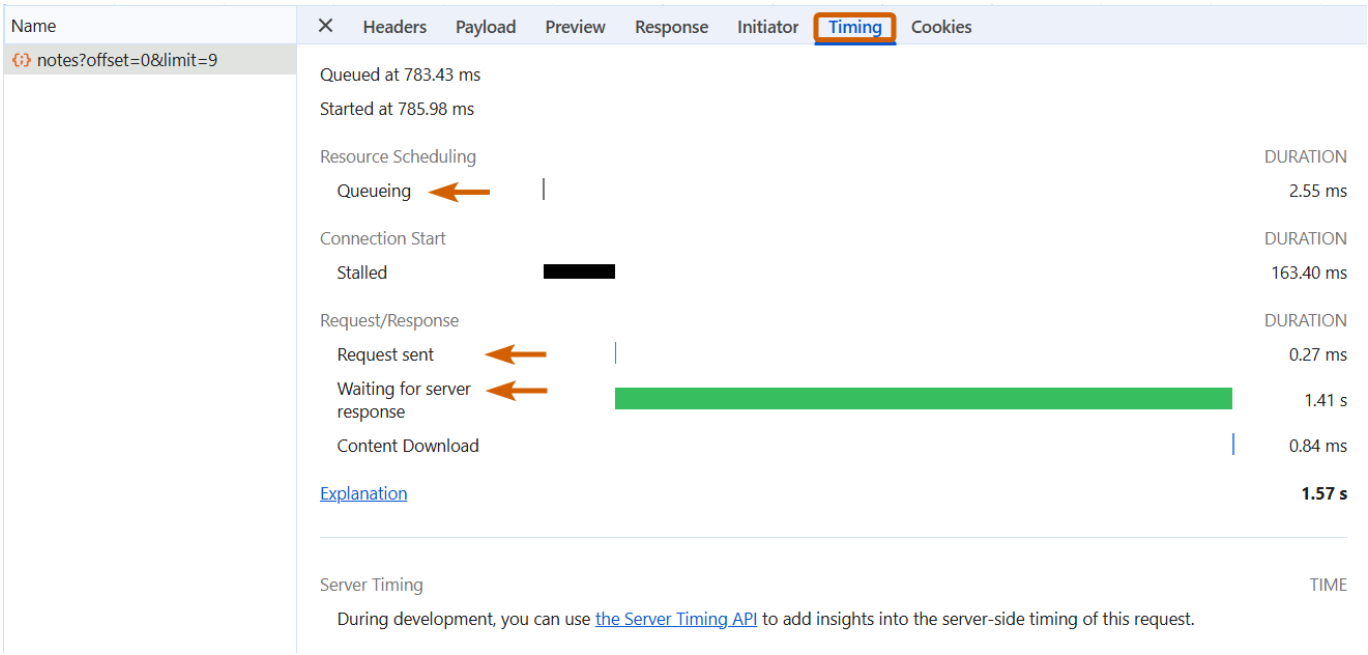
Payload: This tab shows the body of your request, so you can verify if you are sending the correct data.

Preview and Response: The **Response** tab shows the raw data returned from the server. The **Preview** tab provides a more readable, formatted version (e.g., an expandable JSON object or rendered HTML), which is

useful for quickly checking API responses and resources.

Initiator: This tab shows the request call stack and initiator chain, which is useful when trying to identify how and where the resource was called from.

Timing: This tab provides a detailed breakdown of the request lifecycle, showing how much **time** was spent on initial connection, waiting for the server response (TTFB) and content download. This is very useful when a request is slow and you want to know exactly which phase is causing the delay.



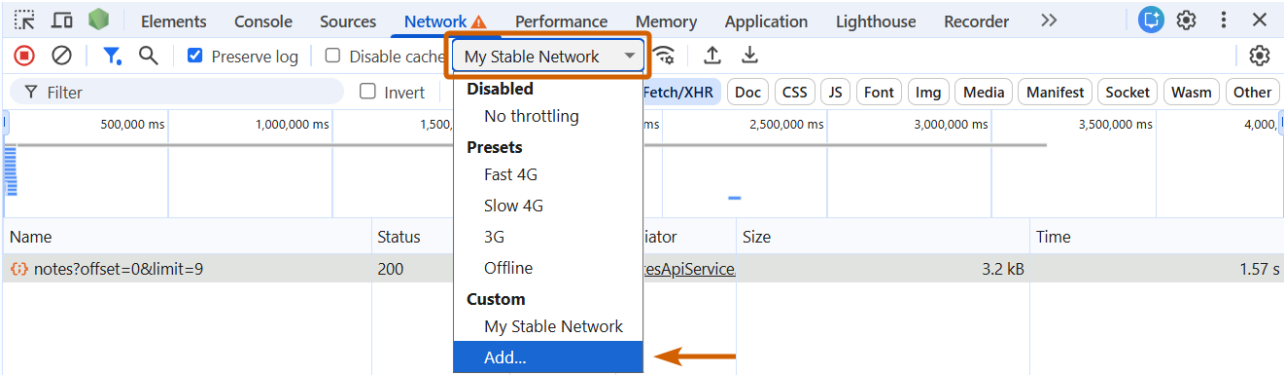
Network Panel: Emulating Network Conditions and Throttling

In real scenarios, users will not have the same high-speed network that you do. So it is important to simulate different network speeds and check how your application performs on slower networks to identify performance bottlenecks and user experience issues.

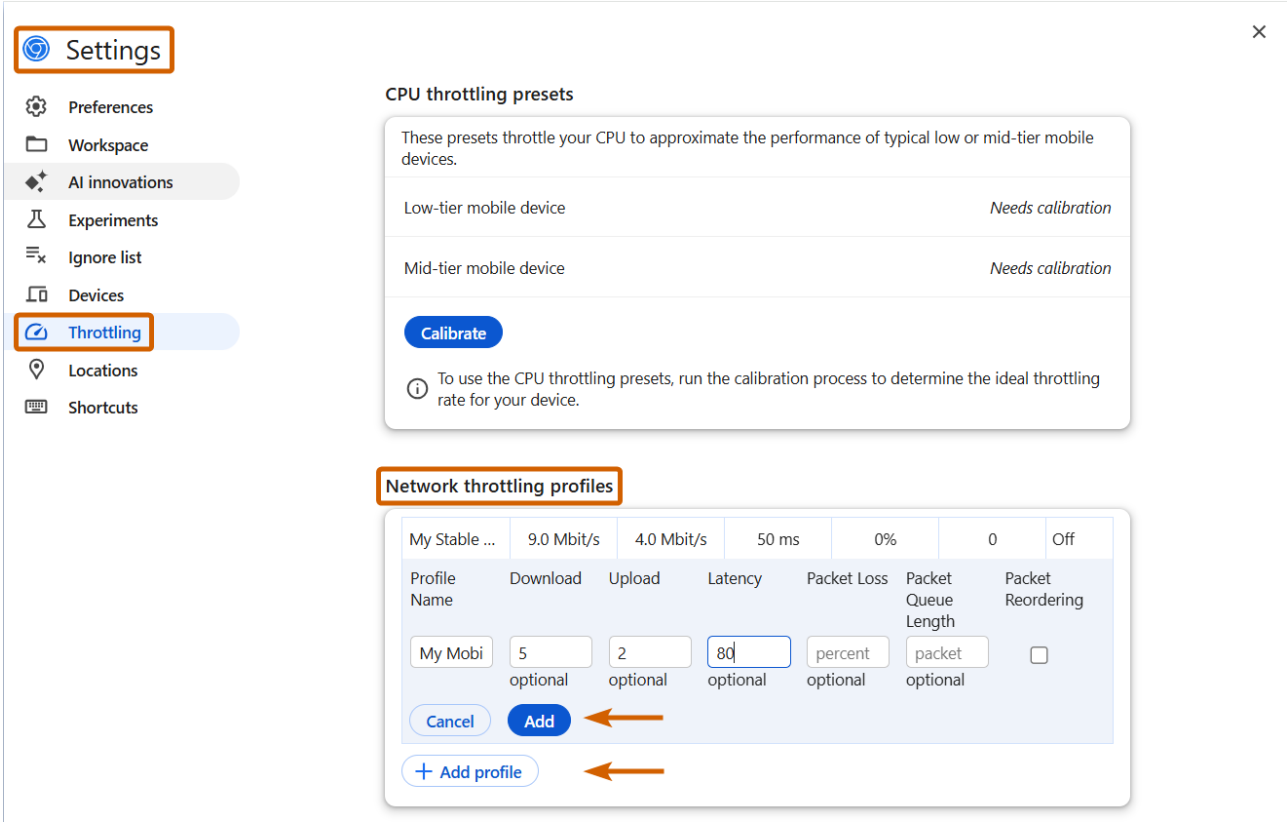
Preset Throttling: At the top of the Network panel, find the dropdown named **No throttling** and select a preset like **Slow 3G** or **Fast 3G**.
Once selected, your application will run at the speed defined by the chosen preset.
You can also use the **Offline** option to simulate a network-unavailable case in the middle of an operation (e.g., an API call) and check how your application responds to it, for example to check whether it displays a clear error message or provides no feedback to the user.

Custom Profiles: You can add your own **Custom Profile** and run your application using it. To do that, follow these steps:

- In the same dropdown, click **Add...**.



- Click **Add profile...** in the **Throttling > Network throttling profiles** section.
- A form will appear where you can enter a custom **Download** speed, **Upload** speed and **Latency**.



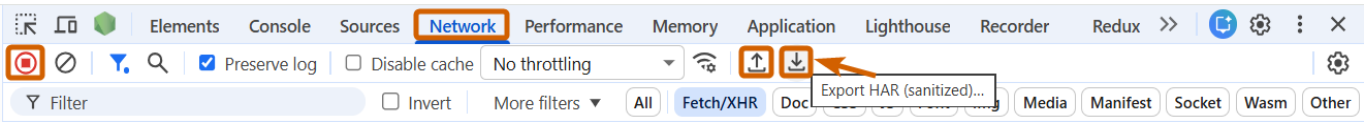
- Click **Add** to save the new profile.
- Close the **Settings** panel. Your new profile will be available under the **Custom** section of the throttling dropdown. You can select it and run your application to test the performance.

Network Panel: Downloading and Importing HAR (HTTP Archive) Files

A **HAR (HTTP Archive)** file is a JSON-formatted log of a web browser's interactions with a website. It records all browser network activity: every request, response, header and sometimes even the response body. Because it can contain sensitive data such as API keys, cookies or user information, you should **always treat HAR files as confidential**.

You can record and download HAR files as well as import them using the **Network** panel. Below are the steps to follow:

- Make sure the red recording icon (⦿) is on. If the icon is grey, click it to start the recording.
- Refresh the page and record the interactions you want to include in the HAR file.
- After finishing, you can stop recording and click the **Export (⭳)** icon to download the HAR file.
- You can then use the **Import (⭱)** icon to import any exported HAR file and debug all network activity within the **Network** panel.



Network Panel: Debugging CORS Errors

Defining CORS policy on the resource server side is an important best practice for the security of your application. It allows you to configure API resources to only be called by legitimate URLs, helping to **prevent external malicious pages reading data from the API**.

CORS-related errors are the most common browser console errors during web development. So it is important to understand how CORS works to be able to debug appropriately.

What is CORS and How It Works

Browsers by default restrict cross-origin HTTP requests made by scripts for security reasons. APIs like `fetch()` and `XMLHttpRequest` follow the **same-origin policy**, which means your application can only request resources from **the same origin it was loaded from**. If your application needs to call a different origin, the other origin must explicitly allow it using **CORS (Cross-Origin Resource Sharing) response headers**.
For example, an application served from `https://note-app-domain.com` cannot fetch data from `https://api-domain.com` unless the API responds with the correct CORS headers, because these are different origins.

Cross-Origin Resource Sharing (CORS) is an HTTP header-based mechanism that lets a server tell the browser which other origins (domain, scheme or port) are allowed to access its resources. The browser applies these rules by checking the server's response headers before making the resource accessible to the client script.

The typical process for a **CORS request** looks like this:

- The browser sends an `Origin` request header to indicate the request's source.
- The **simple requests** (e.g., `GET`, `POST` or `HEAD` with specific conditions) are sent directly: the browser checks for a valid `Access-Control-Allow-Origin` in the response.
- For **preflight** requests (like `PUT`, `DELETE` or with custom headers), the browser makes a separate OPTIONS request first to check if the server hosting the cross-origin resource permits the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers

that will be used in the actual request. The server must return valid CORS headers if the request is allowed.
- If the origin and response headers match, the browser grants access, otherwise, it blocks the response and **throws CORS error in Console**.

Below are key response headers involved in CORS:

- `Access-Control-Allow-Origin`: Indicates whether a resource can be accessed by web pages from a different origin.
- `Access-Control-Allow-Credentials`: Indicates whether the server allows credentials (e.g., cookies, authentication headers) to be included in cross-origin HTTP requests.
- `Access-Control-Allow-Methods`: Indicates one or more HTTP request methods allowed when accessing a resource.
- `Access-Control-Allow-Headers`: Indicates the HTTP headers that can be used during the actual request.

Not matching the origin and server response headers policies for any of these headers will result in a **CORS error** on the browser side.

CORS Policy Configuration Example

Below is an example of simple **Express.js** service **CORS policy** configuration to start with:

```
let express = require('express');
const cors = require('cors');
const app = express();

app.use(cors({
    origin: [
        'http://local.react-note-app.com:3000',
        'https://note-app-domain.com',
        'http://localhost:3000',
    ], // Specify allowed domains
    methods: 'GET, POST, PUT', // Specify allowed methods
    allowedHeaders: 'Content-Type, Authorization', // Specify allowed headers
    credentials: true, // Allow cookies/auth headers
}));

app.use(express.json());
```

Access-Control-Allow-Origin Header

`Access-Control-Allow-Origin` is the fundamental response header in Cross-Origin Resource Sharing (CORS) that indicates whether a resource can be accessed by web pages from a different origin.

Below are its allowed values:

- *: Permits access from any origin. This is suitable for public APIs but unsafe for requests involving credentials.
- **Specific origin** (e.g., `http://local.react-note-app.com:3000`): Grants access only to the listed origin. This header can contain only one origin value. In practice, servers often return the request's `Origin` value when it is allowed by the server's CORS policy.
- **null**: Used in sandboxed environments such as file URLs or data URLs.

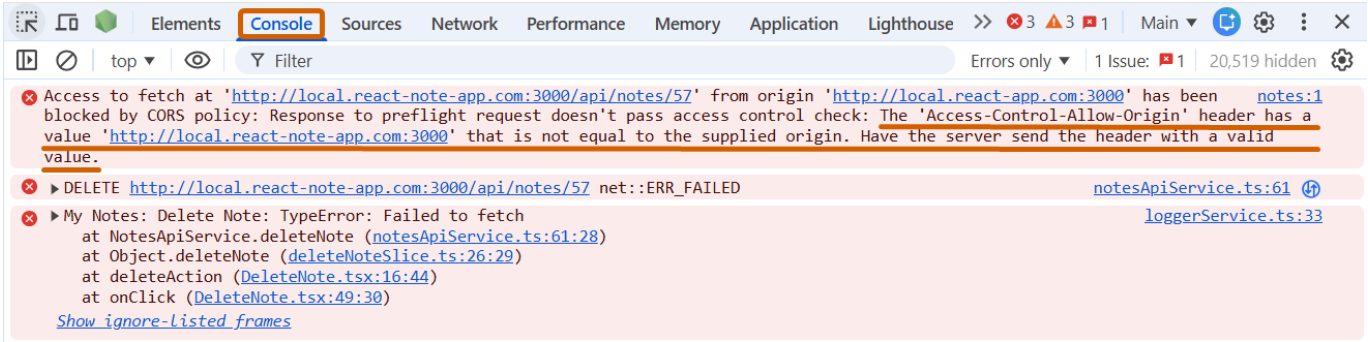
If a server doesn't send this header, browsers automatically block cross-origin requests, resulting in a **CORS error**.

CORS Policy Debugging Tips

Below are debugging tips for CORS policy:

- Watch out for `Access-Control-Allow-Origin` response header.
  - If it uses `*` value and your API is not public, it is considered a **red flag**. Permitting any origin to access a non-public API breaks your application security.
  - Check this value on **production environment**. If it permits a **local URL to access a production resource** (e.g., `http://local.react-note-app.com:3000`), it is also a **red flag**.
    The private API can give access to its resources on testing or even staging environments for local development tests. These environments often run under VPN and always have PII (Personally Identifiable Information) data masked so there is no security risk. However, the private API should not permit local access to its production URL, as this environment contains real user data.
- If you notice CORS errors:
  - Check the CORS policy defined on server side, it may be broken, not configured or just miss the origin from which the web application request is made.
  - Also make sure to check CORS error reason which is provided in the Console error message by the browser. In many cases it describes very well the root cause of the issue.

For example, in this case the root cause is that the `Access-Control-Allow-Origin` header in the preflight request response was different from the expected origin value. So the browser blocked the response with a CORS error.
The solution is to add `http://local.react-note-app.com:3000` URL in the allowed origin list on the server side.



Network Panel: Debugging Caching Headers

Caching Headers

`Cache-Control`, `ETag` and `Last-Modified` are important headers for controlling the cache of your web application.

- **Cache-Control:** An HTTP header that dictates browser caching behavior. It sets the rules which determine whether the user will have resources loaded from their local cache, or whether the browser must send a request to the server for fresh resources. Below are the most commonly used values for this header:
  - **max-age=N**: The response remains fresh until `N` seconds after the response is generated.
  - **must-revalidate**: The response can be stored in caches and can be reused while fresh. If the response becomes stale, it must be validated with the origin server before reuse.
  - **no-cache**: The response can be stored in caches, but the response must be validated with the origin server before each reuse.
  - **no-store**: Indicates that any caches of any kind (private or shared) should not store this response.
    If a page is using this header, it is a strong directive to the browser not to store any data in the cache, including bfcache. This option should only be used for pages that contain personal or private information. For other pages, to keep content reasonably fresh, use `Cache-Control: no-cache`, or a short cache time like `max-age=60` to benefit from caching.
  - **private**: Indicates that the response can be stored only in a private cache (e.g., local caches in browsers).
    You should add the **private** directive for user-personalized content, especially for responses received after login and for sessions managed via cookies.
  - **public**: Indicates that the response can be stored in a shared cache (e.g., browsers, CDNs, etc). Responses for requests with Authorization header fields must not be stored in a shared cache.

  For example, the following config for static resources means that the resource is stored in a public cache, is considered fresh for a month, after which it must be revalidated with the server.

  ```
  Cache-Control: public, max-age=2592000, must-revalidate
  ```

- **ETag (Entity Tag):** An HTTP response header acting as a unique identifier or **fingerprint** for a specific version of a web resource, used by servers and browsers for efficient caching. The typical use case of the ETag header is to cache resources that are unchanged. Below is the process:
  - **Server Sends `ETag`:** When a client makes an initial request for a resource, the server generates a unique identifier and sends it in the response's `ETag` header, along with the resource itself. The server also ensures that `ETags` are regenerated whenever the resource is modified.
  - **Client Stores `ETag`:** The client (e.g., a web browser) caches the resource and its associated ETag value.

- **Client Sends `If-None-Match`:** On subsequent requests for the same resource (once the cache is stale), the client includes the previously stored `ETag` value in the `If-None-Match` request header: `If-None-Match: W/"adb-tdszMNMqWVopvDR4KDdqtgK/CrI"`
- **Server Validates:** The server receives the `If-None-Match` header and compares the provided `ETag` with the `ETag` of the current version of the resource on the server.
  If they match, the resource has not changed. The server sends a `304 Not Modified` status code with an empty body, telling the client to use its cached version.
  If they do not match, the resource has changed. The server sends the new resource with a `200 OK` status and a new `ETag` header value, and the client updates its cache.

In some frameworks, `ETag` header generation is controlled automatically like in the case of **Next.js**. In other cases, you need to configure it in server settings and also make sure that ETag is regenerated with each resource update. In most modern web browsers, sending the `If-None-Match` header is handled automatically by the browser's built-in caching mechanisms, though some custom client implementations might require you to write custom code to manually send `If-None-Match` header from the client-side.

- **Last-Modified:** This header contains a date and time when the origin server believes the resource was last modified.
  This header serves the same purpose as `ETag` but uses a **time-based** strategy to determine if a resource has changed, as opposed to the **content-based** strategy of `ETag`. Similar to `ETag`, the client stores the `Last-Modified` header value on initial load. On subsequent requests, it sends this value in the `If-Modified-Since` header, allowing the server to decide whether to send a new resource or a `304 Not Modified` response.
  Due to its nature, `Last-Modified` is a weaker header than `ETag`. It can be set as a fallback for `ETag`, for example by setting it during each resource build. If both are present, the server will give priority to the `ETag` header.

Resource Caching Best Practices

Caching best practices highly depend on individual resource type and essence of the application. Below are some general best practices to consider when debugging resource cache:

- **Define `ETag` or `Last-Modified` headers**: This helps to revalidate resources and use the cached version for a long time if it is still fresh. `ETag` usually is more accurate.

- **Attach versioning to static resources after each update**: This can be done through concatenating a unique identifier (e.g., current timestamp) to the resource name (e.g., `src_app_layout_tsx_78cdd4a3._.js`) or adding unique GET parameter (`src_app_layout.js?v=1767976737453`). Both methods ensure that regardless of the cache policy, the fresh copy is served to the user if the file is changed, because the browser treats these modifications as new files.
  The bundlers used by modern frameworks like **Next.js** and **Angular** already attach unique chunk names to the generated files, so in most cases you don't have to worry about it.

- **Define `Cache-Control` header**: By default this header can have different values based on the server environment and may not be suitable for your application goals. So the best option is to always define

it per resource:

- For Secure APIs (e.g., financial, documents), use:

```
// Don't cache at all
Cache-Control: no-store
```

  This ensures that no secure information is stored on user's physical device and protects data from malicious attacks.

- For other user specific APIs (e.g., notes, preferences), use:

```
// Cache but revalidate for each request
Cache-Control: private, no-cache

// This is equivalent to
Cache-Control: max-age=0, must-revalidate
```

  This ensures the browser always revalidates for a fresh copy but at the same time benefits from `ETag`.

- For public APIs, use:

```
// Cache for a month
Cache-Control: public, max-age=2592000
```

  Adjust `max-age` with the frequency that you deploy API changes (e.g., a month, a week).
  You need to specify `max-age` time carefully because `Cache-Control` has priority over `ETag` and the request revalidation will not be sent to server if the resource is considered fresh by `Cache-Control` header.

- For versioned CSS/JavaScript files, images or fonts, use:

```
// Cache for a year, CDN-friendly
Cache-Control: public, max-age=31536000, immutable
```

  The immutable response directive indicates that the response will not be updated while it's fresh. This is the best choice for bundled versioned files and ensures they are not validated each time.

- For non-versioned static CSS/JavaScript files and non-secure, non-personalized public HTML files, use:

```
// Cache for shorter duration, 5 minutes
Cache-Control: public, max-age=300, must-revalidate
```

- For secure authenticated application HTML files, use:

```
// Cache for shorter duration, 5 minutes, private cache
Cache-Control: private, max-age=300, must-revalidate
```

Network Panel: Debugging Access Tokens

In order to understand if the authentication bearer token sent to the API is valid or not, let's explore different types of security tokens.

Security Token Types

There are two types of security tokens:

- **ID Tokens**: These are user **authentication** tokens provided by the **OpenID Connect** (**OIDC**) protocol (an open standard for decentralized authentication). The typical workflow looks like this: users authenticate themselves (e.g., by providing credentials during login) and as a result OIDC provides an ID token that proves the user has been authenticated.
  - These tokens are specially encoded as **JWT (JSON Web Token)**. You can use jwt.io to decode them.
  - ID tokens typically contain **user PII** such as name, email and address and are intended to be extracted by **the client application** for getting necessary user information to show in the UI (e.g., in Profile page).
  - The main mistake developers make is using ID tokens for API authorization, which is wrong because these types of tokens do not have any authorization information and **should not be sent to an API**.
- **Access Tokens**: These tokens are specifically designed for **authorization** purposes, allowing access to a resource (e.g., `file`, `API`) on **the server side**. Access tokens come from **OAuth 2.0** which is a protocol designed to access specific resources on behalf of the user. This is what is known as a delegated authorization scenario, where the user delegates a client application to access a resource on their behalf through an access token.
  - These tokens are defined by **scopes** which describe which resources the token has access to (`notes.create`, `notes.delete`, `notes.read`).

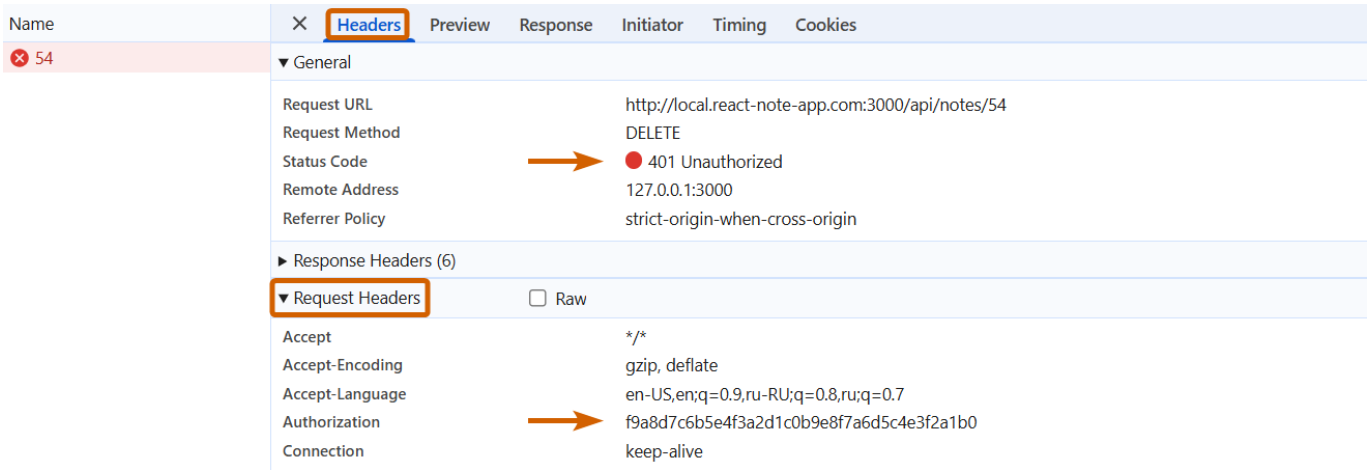
- They are **not required to be JWTs** and can have any string format, but in most cases they are implemented as a JWT.
- They are **meant to be sent to the resource server to access some sort of data**, so these are the tokens **used as API authentication bearer tokens**.
- Access tokens **are not used for user authentication** because using an access token, you cannot make any assumption about user identity or about whether they are logged in or not.

Access Tokens Debugging Tips

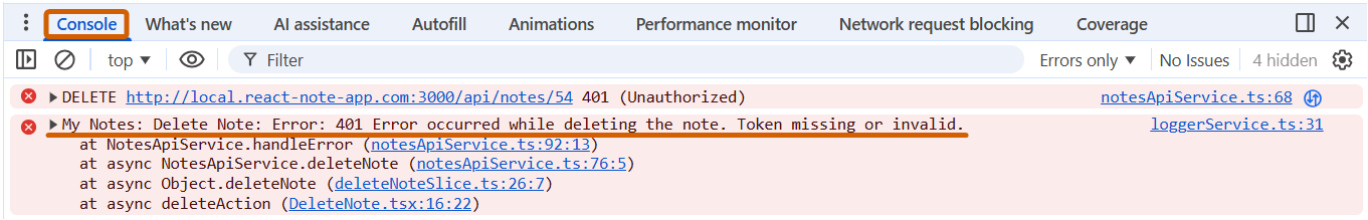
The main indicator of a bearer (access) token issue is the `401 Unauthorized` error on the client side. Below are tips for debugging that error:

- Check for the error message printed in the **Console**. In many cases it may contain useful information to identify the root cause of the issue. As a best practice, normally client applications log the server error message in the Console for more information.
- Check if the `Authorization` header is passed to the API in **Request Headers**.
- Check if the bearer (access) token is sent appropriately in the `Authorization` header. It could be missing, empty or `null`.
- Check if `Bearer` text is included in the token. This is one of the most common mistakes developers make.
  Correct example looks like this: `Bearer f9a8d7c6b5e4f3a2d1c0b9e8f7a6d5c4e3f2a1b0`
- Check if the token passed to the API contains PII. While it depends on the company **IAM (Identity and Access Management)** setup, in general, if you see a token containing PII passed to the API, it is likely an **ID token**, which is not intended for sending to the back-end. For JWTs, you can use jwt.io to parse the token to get more information from it.
- If the token is a valid access token, check for the expiration date as it might be just expired.

For example, in this case the `DELETE` request failed because `Bearer` text was not passed to the API in the `Authorization` request header.



Consequently, the client application prints the server error message, which helps easily debug the issue and understand the root cause.



Network Panel: Common HTTP Status Codes: Cheat Sheet

The HTTP status code is usually a strong indicator helping to understand the API request outcome, particularly the root cause of the API request failure. Below is a cheat sheet of common HTTP status codes appearing for business APIs on the client side.

1. Successful Responses (2xx)

These codes indicate that the action requested by the client was received, understood and accepted.

Code	Status	Exact Explanation
200	OK	The request succeeded. The result and meaning of **success** depend on the HTTP method: - **GET:** The resource has been fetched and transmitted in the message body. - **HEAD:** Representation headers are included in the response without any message body. - **PUT or POST:** The resource describing the result of the action is transmitted in the message body. - **TRACE:** The message body contains the request as received by the server.
201	Created	The request succeeded, and a new resource was created as a result. This is typically the response sent after `POST` requests or some `PUT` requests.
202	Accepted	The request has been received but not yet acted upon. It is intended for cases where another process or server handles the request, or for batch processing.
204	No Content	There is no content to send for this request, but the headers are useful. The user agent may update its cached headers for this resource with the new ones.

2. Redirection Messages (3xx)

These codes indicate that the client must take additional action to complete the request.

Code	Status	Exact Explanation
301	Moved Permanently	The URL of the requested resource has been changed permanently. The new URL is given in the response.
302	Found	The URL of the requested resource has been changed temporarily. Further changes in the URL might be made in the future, so the same URL should be used by the client in future requests.

Code	Status	Exact Explanation
304	Not Modified	This is used for caching purposes. It tells the client that the response has not been modified, so the client can continue to use the same cached version of the response.

3. Client Error Responses (4xx)

These codes indicate that the error originated from the client (the browser or your code).

Code	Status	Exact Explanation
400	Bad Request	The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing or deceptive request routing).
401	Unauthorized	Although the HTTP standard specifies **unauthorized**, semantically this response means **unauthenticated**. That is, the client must authenticate itself to get the requested response. This often happens when the bearer token passed to the API is missing or is expired.
403	Forbidden	The client does not have access rights to the content. It means the client is unauthorized, so the server is refusing to give the requested resource. Unlike `401 Unauthorized`, the client's identity is known to the server. For example, the client does not have permission to delete or edit a resource.
404	Not Found	The server cannot find the requested resource. In the browser, this means the URL is not recognized. In an API, this can also mean that the endpoint is valid but the resource itself does not exist. Servers may also send this response instead of `403 Forbidden` to hide the existence of a resource from an unauthorized client.
405	Method Not Allowed	The request method is known by the server but is not supported by the target resource. For example, an API may not allow `DELETE` on a resource, or the `TRACE` method entirely.
409	Conflict	This response is sent when a request conflicts with the current state of the server. 409 responses are errors often sent to the client so that a user might be able to resolve a conflict and resubmit the request.
422	Unprocessable Content	The request was well-formed but was unable to be followed due to semantic errors.
429	Too Many Requests	The user has sent too many requests in a given amount of time (rate limiting).

4. Server Error Responses (5xx)

These codes indicate the server failed to fulfill an apparently valid request.

Code	Status	Exact Explanation
500	**Internal Server Error**	The server has encountered a situation it does not know how to handle. This error is generic, indicating that the server cannot find a more appropriate 5xx status code to respond with.
502	**Bad Gateway**	This error response means that the server received an invalid response while working as a gateway to get a response needed to handle the request.
503	**Service Unavailable**	The server is not ready to handle the request. Common causes are a server that is down for maintenance or that is overloaded. Note that together with this response, a user-friendly page explaining the problem should be sent.
504	**Gateway Timeout**	This error response is given when the server is acting as a gateway and cannot get a response in time.

Network Panel: Common Use Cases

- **Debugging API Requests**: The **Network** panel is the primary tool for diagnosing failed API calls. You can:
    - Inspect the **Payload** to ensure the correct data is being sent.
    - Check **Request Headers** for missing or incorrect authentication tokens (e.g., Bearer tokens).
    - Verify **Response Headers** for correct CORS policies (e.g., `Access-Control-Allow-Origin`).
    - Preview the **Response** to see if the data model matches what the front-end expects.
- **Analyzing Resource Load and Caching**:
    - Quickly see how long each asset (`CSS`, `JS`, `images`) takes to load, identify the largest files and verify whether resources are being served from the **browser cache** (`disk cache` or `memory cache`) instead of being re-downloaded.
    - Check the `Disable Cache` checkbox to completely prevent the browser from caching resources, allowing you to do effective local development and debug your application with fresh resources each time.
    - Verify caching headers (e.g., `ETag`, `If-None-Match` and `Cache-Control`) to see if resource caching is enabled appropriately.
- **Diagnosing Request Latency**: Use the **Timing** tab for any request to get a detailed breakdown of its lifecycle. This helps you identify whether a delay is caused by a slow DNS lookup, server wait time (Time to First Byte) or content download.
- **Simulating Network Conditions**: Use the **throttling** dropdown to simulate slow connections (e.g., `Slow 3G`) to test how your application behaves for users on slower networks.
    - Slow networks are also a great way to check the effectiveness of loading indicators, skeletons and thumbnails.

    - Offline network simulation can help you verify the user experience in failure scenarios and ensure that users receive a constructive error message instead of being left confused.

- **Filtering and Searching Requests**: Use the **Filter** bar to quickly find specific requests by name, domain or type (e.g., `Fetch/XHR`, `Img`, `CSS`), which is essential for debugging complex pages with many resources.
- **Export/Import Network Activity**: Use **Import and Export HAR (HTTP Archive)** features to record the full network activity of a specific bug scenario and include it in the QA ticket to give peers better context to debug the underlying issue.

> **Note:** During the debugging process it is often useful to block some requests in order to remove noise and debug issues more accurately. For this purpose you can use the **Network request blocking** panel in DevTools.
> This panel lets you block multiple resources or patterns at the same time. When it is active, you can also right-click on any request from the **Network** panel and choose **Block request domain** or **Block request URL** to block specific domains or URLs. The blocked patterns will then appear in the **Network request blocking** panel where you can edit or remove them.
>
> **How to open:** Open DevTools and run the **Show Network request blocking** command from the **Command Menu**.

1.7 Application Panel

The **Application** panel is your window into everything the browser stores locally for a web page. It is the primary tool for debugging **state**, **caching** and **offline** functionality of your application, allowing you to inspect and manage storage, service workers and application manifest data.

We will explore the following three main sections of the Application panel:

- **Application**
- **Storage**
- **Background services**

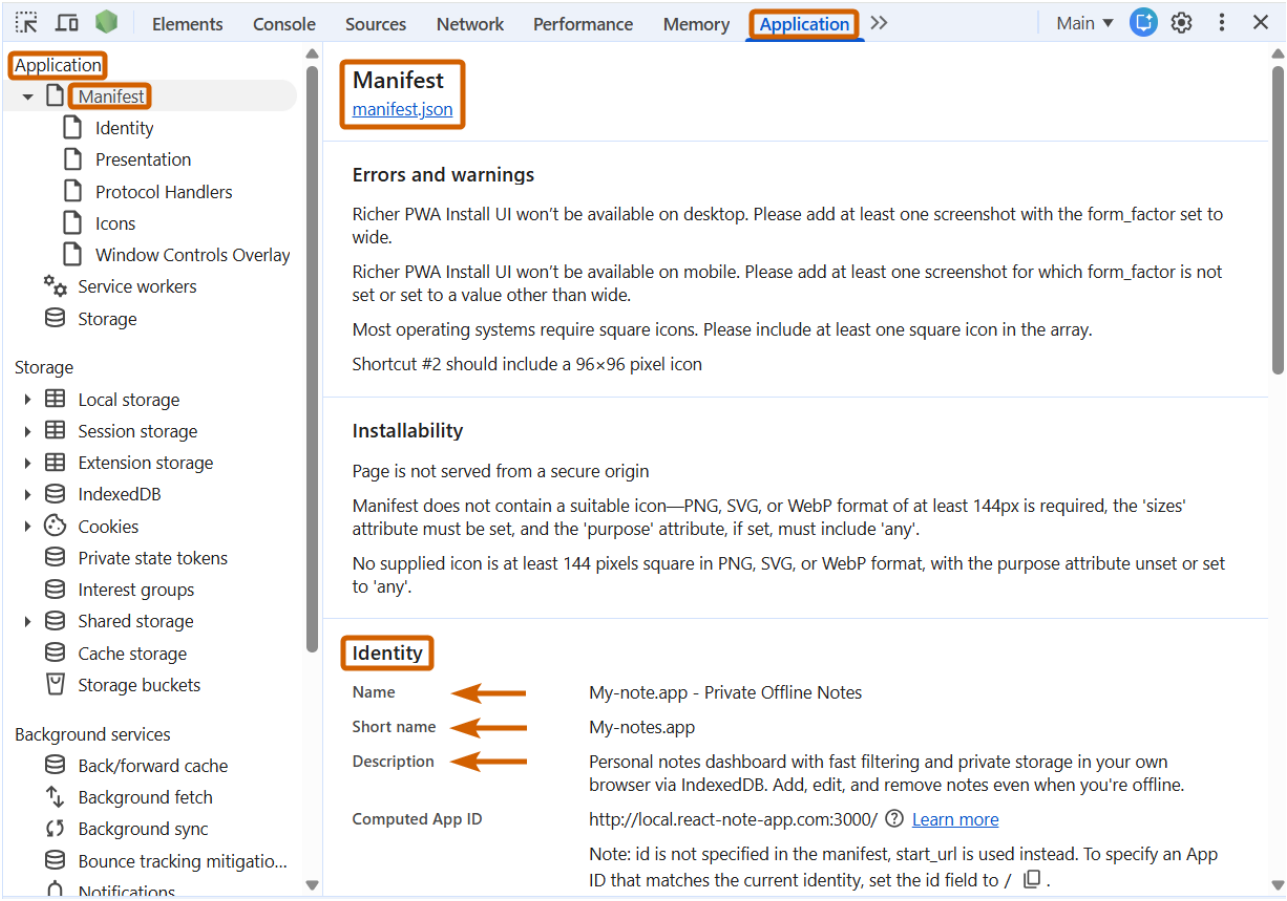
Application Panel: How to Open

To open the **Application** panel, open DevTools and click the **Application** tab, or select it from the **Command Menu**.

Application Panel: Application Section

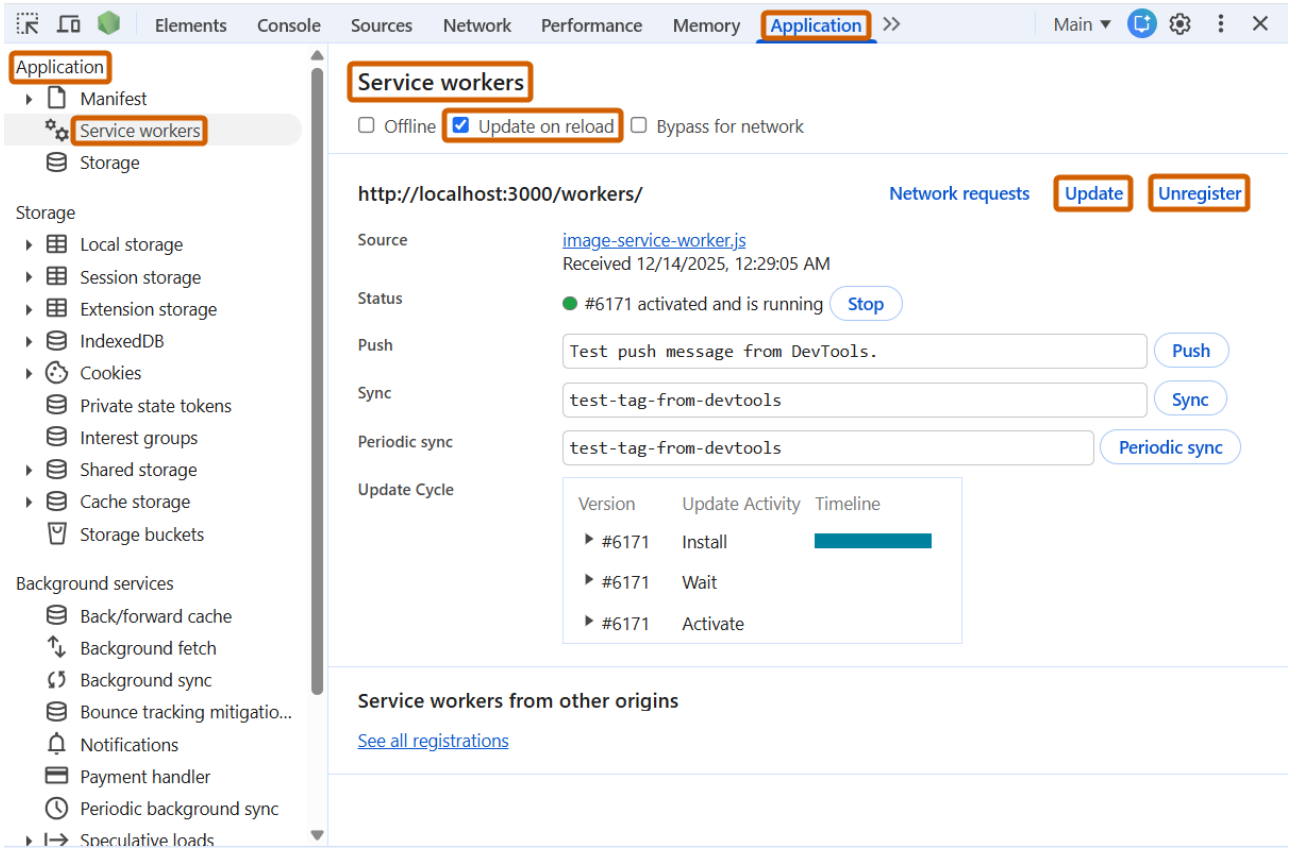
This section shows high-level information about how your application behaves. It is divided into the following tabs:

- **Manifest:** Presents information from your web application's `manifest.json` in a user-friendly way. It also displays errors and warnings, if any, in the corresponding sections.



- **A web application manifest** (`manifest.json`), defined in the Web Application Manifest specification, is a `JSON` text file that provides information about your web application.
- Web application manifests are referenced in your HTML pages using a `<link>` element in the `<head>` of a document: `<link rel="manifest" href="manifest.json" />`
- The most common use for a web application manifest is to provide information that the browser needs to install a **Progressive Web App (PWA)** on a device, such as the application's name and icon.

- **Service workers:** Allows you to inspect and debug service workers by checking their source file names, installation status, emulating push events, updating/unregistering a service worker and more.
  - **Web Worker** itself is a feature in web browsers that allows JavaScript code to run in a background thread, separate from the main browser thread, thus creating the concept of true parallelism (multithreading).
  - **A Service Worker** is a special type of Web Worker that essentially acts as a proxy server sitting between web applications, the browser and the network (when available). It is intended to enable **effective offline experiences**, intercept network requests to handle connectivity changes and update cached assets from the server. It also allows access to **push notifications** and **background sync APIs**. Basically, service workers help make web applications behave more like native applications.
  - Service workers work with secure HTTPS protocol, and locally they may not work if you run them by domain with HTTP protocol, such as `http://local.react-note-app.com:3000`. For local development, you may need to use localhost in your URL in order to get service-workers loaded: `http://localhost:3000`.

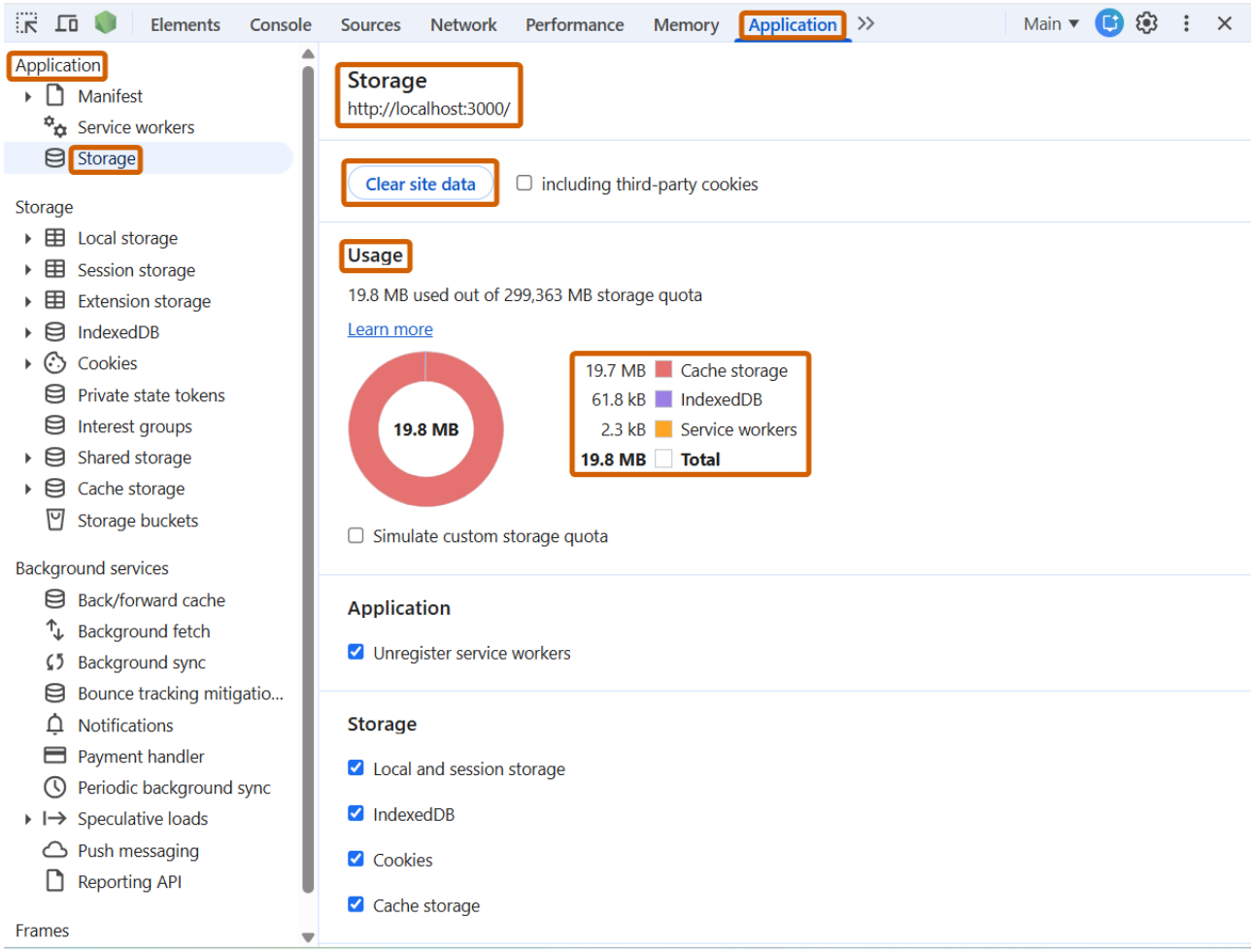


Useful Debugging Tips

- One useful debugging tip of this tab is that if there are new changes in the service worker file but the file is cached, you can click **Update** to run the installation again.
- During local development you may want to keep **Update on reload** button checked so that every time you do new changes and reload your app, the changes take effect.
- Or alternatively, if you accidentally removed service worker's cache data from **Storage > Cache Storage**, you can click **Unregister** to unregister it, then reload your application to register it again by running your application.

If there are any installed service workers, they will look like this in the Service Workers tab:

- **Storage:** Provides a high-level overview and powerful tools for managing all forms of client-side storage.
  - **Check Storage Size**: Get a breakdown of how much data is being used by each storage type (e.g., **Local Storage**, **Session Storage**, **IndexedDB**, **Cache**). This is useful for ensuring your application isn't consuming excessive disk space on a user's device.
  - **Clear Site Data**: The **Clear site data** button is invaluable for debugging. It allows you to completely reset the application's state by clearing out all storage, caches and service workers. This is useful for testing the **first-load** experience for new users.



Application Panel: Storage Section

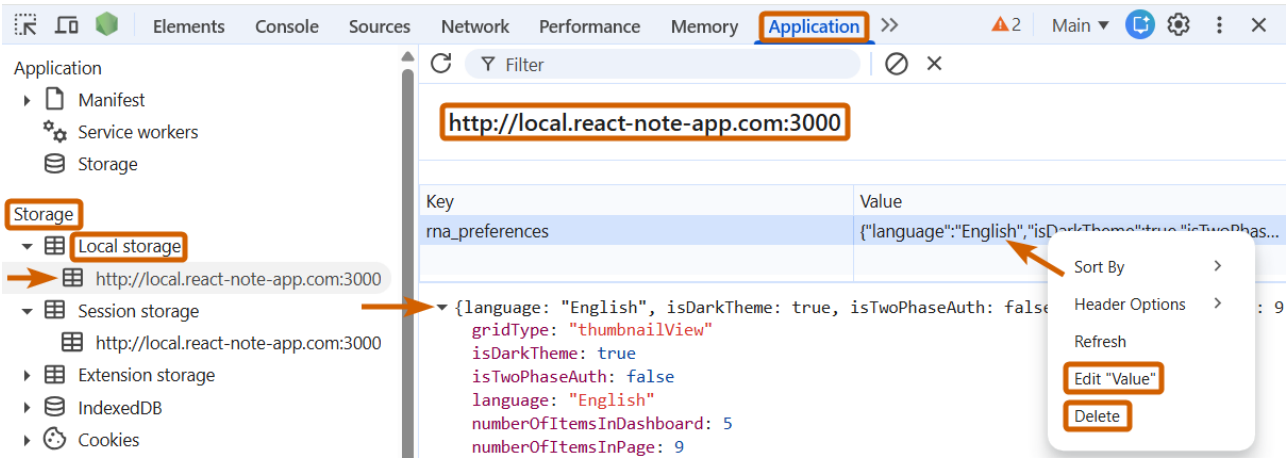
Local Storage and Session Storage Tabs

`localStorage` allows websites to store key-value pairs persistently in a user's browser, even after the browser is closed and reopened. `sessionStorage` is similar to `localStorage`, except that while `localStorage` data has no expiration time, `sessionStorage` data gets cleared when the page session (tab/window) ends, that is, when the page is closed.
Local storage and **Session storage** tabs are very useful for checking client-side data persistence. Below are some useful functionalities:

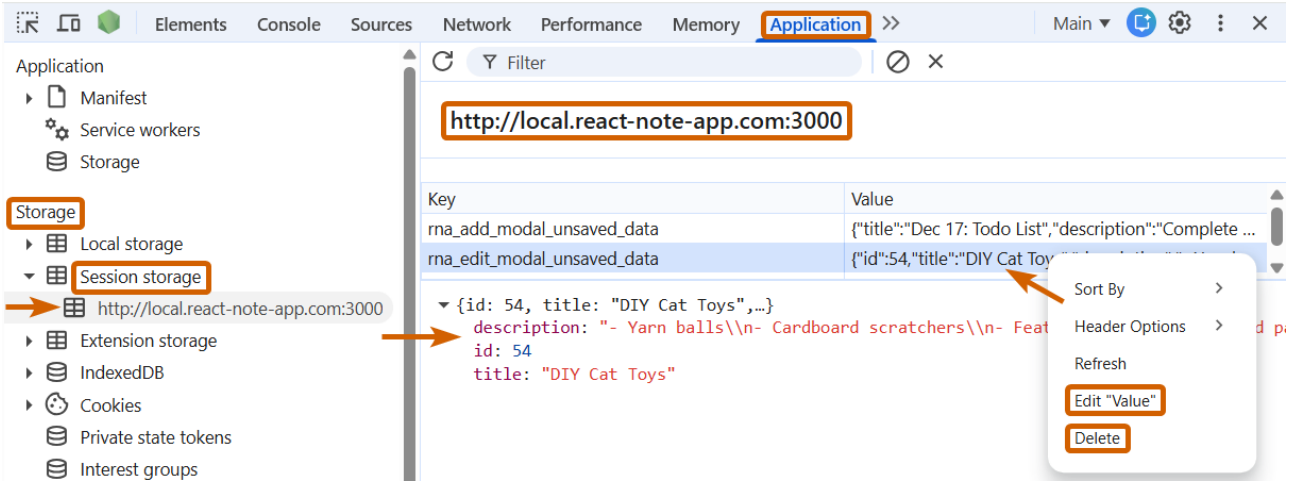
- **Inspect Data**: View the key-value pairs your project and any third-party tools (like analytics or authentication libraries) are storing.
- **Live Editing**: You can **add**, **edit** or **delete** entries on the fly to test how your application reacts to different stored states without writing any code.
    - To **edit** and **delete** an entry, right-click on the entry and choose **Edit "Value"** and **Delete** options respectively.
    - To **add** a new entry, double-click the next available row and type the **Key** and **Value**. The new entry will be saved automatically when you press `Enter` or click outside.

Local Storage and Session Storage: Use Cases

- Use **Local storage** to store data which should be persistent across multiple sessions or after the browser is closed, such as **user application settings and preferences** or **temporary cached data** of your application if it is not too big (max `5 MB`):



- Use **Session storage** to store data which should be removed after session is completed (browser tab is closed), such as **unsaved modal information**, **expanded subsections**, **selected categories**, etc.



Cookies Tab

Cookies tab lets you inspect all cookies associated with the current domain. A cookie is similar to Local Storage / Session Storage: it also stores data as key-value pairs in the browser. However, there are important differences:

- Cookies (`4 KB`) are much smaller than Session Storage / Local Storage (`5 MB`).
- Cookies are based on domain, and you can set up a cookie which can be shared across subdomains while session/local storage are specific to the subdomains where they were created.
- Cookies are sent with HTTP requests to the back-end automatically for same-origin requests or via the `credentials`/`withCredentials` setting for cross-origin requests. Therefore, they are primarily used to store data that is needed for back-end communication.

This tab provides useful functionalities to debug and work with cookies:

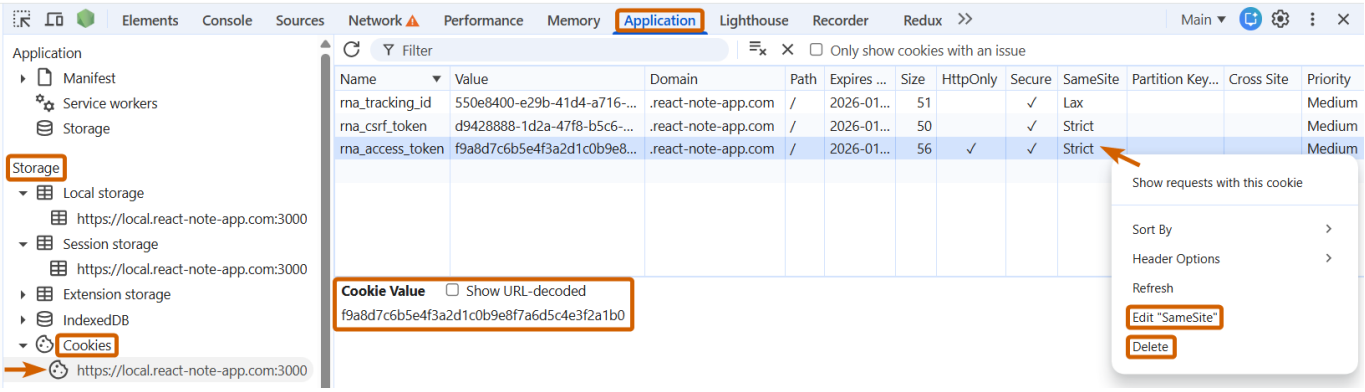
- **Identify Conflicts**: You can easily see all cookies at once, which helps identify **cookie name conflicts** where different scripts might be overwriting each other's values.
- **Check Attributes**: Verify critical security and behavior attributes needed for debugging authentication and tracking issues for any cookie, such as:
  - `HttpOnly`: Cookie can't be read or modified by JavaScript (e.g., via `document.cookie`). It is only sent in `HTTP` requests and can only be accessed when it reaches the server, which helps to protect against XSS stealing session cookies. The `HttpOnly` flag is assigned to the cookie **by the server only, not the client**.
  - `Secure`: Cookie is sent only over `HTTPS` (not over plain `HTTP`).
  - `SameSite`: Controls whether the cookie is sent on cross-site requests:
    - `Strict`: Only same-site requests
    - `Lax`: Same-site requests plus top-level navigations
    - `None`: Cross-site requests allowed (must also be `Secure`)
  - `Expires`: Shows absolute date and time when the cookie is removed. If not set, it is considered a session cookie and is deleted when the browser session ends.
- **Clear and Modify**: You can modify or delete cookies individually or clear all of them for the application to test login or tracking flows.

Cookies are typically used for small data that needs to be sent to the back-end or needs to be available across subdomains, like **authentication access tokens**, **user-specific preferences** (saved language settings, UI theme type, layout choices) or **tracking analytics keys** (shared across subdomains).

For example, the secure protection of your API requests may look like this:

- Setting **API access token** cookie from the server as `HttpOnly`, `SameSite: Strict` and `Secure` with session expiration date.
  This helps to protect against **XSS attacks stealing session cookies** because the token cannot be read from the client-side.

- Also setting **CSRF token cookie** from back-end for each session as `Secure` and `SameSite: Strict`.
  The front-end can access this cookie and send as `X-CSRF-TOKEN` header for each API request which will prove to the server the front-end fingerprint, meaning that the request was sent from the correct client application.
  This helps to prevent **Cross-Site Request Forgery (CSRF) attacks** because if another domain sends a request to your APIs, it will be rejected on server side due to missing CSRF token which is available only for the same origin domains.

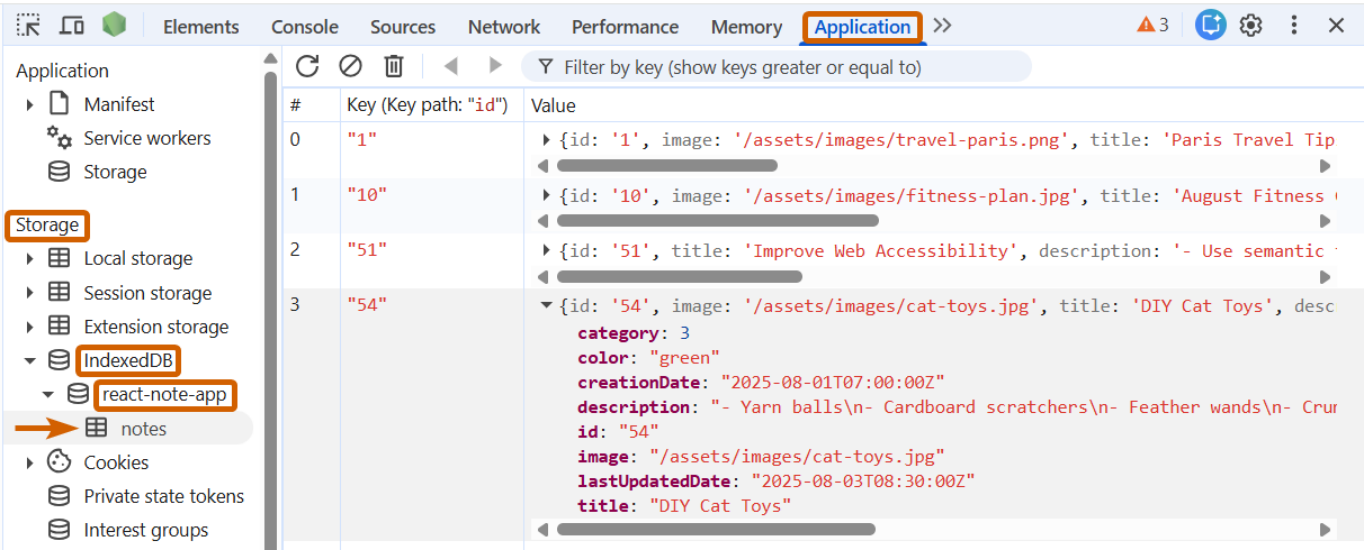


IndexedDB Tab

IndexedDB tab lets you **inspect, delete and refresh client-side object stores (databases)** from the browser. IndexedDB is a powerful, low-level client-side database built into browsers, allowing web applications to store large amounts of structured data (like files/blobs, complex objects, JSON) locally on the user's machine.

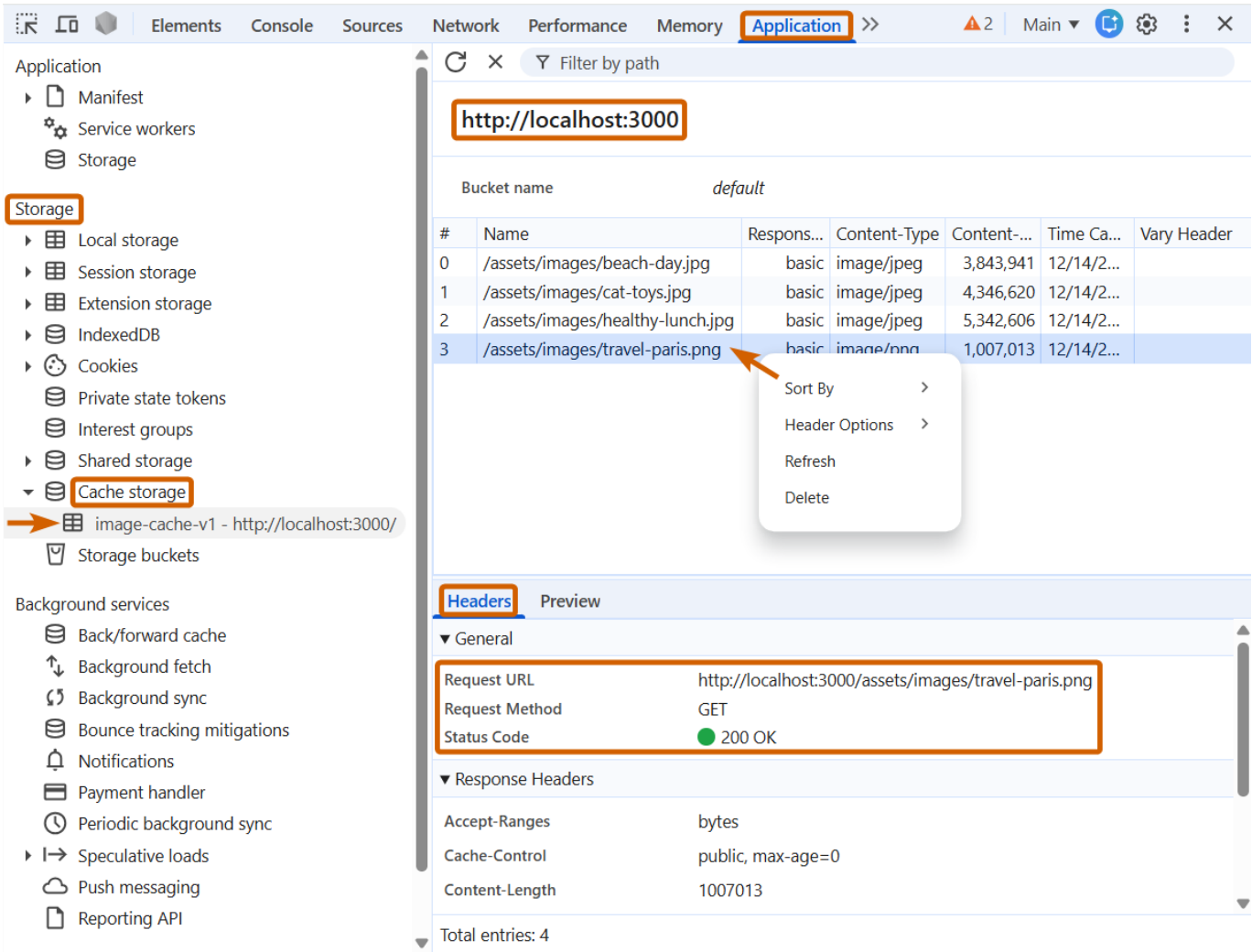
- It enables **offline functionality and rich querying**, similar to a NoSQL database with transactional support for data integrity. Because it lets you create web applications with rich query abilities regardless of network availability, your applications can work both online and offline.
  For example, you can allow users to work offline in a Note application and store the data on IndexedDB in the browser, then a separate service worker job can periodically sync the changed notes with the database.
- It is an **effective mechanism for client-side caching**. IndexedDB can be used for storing various assets from complete sets of customer records to even complex data types like audio or video files.
  For example, you could download a batch of music files (perhaps used by a web game or music player application), store them inside a client-side database and play them as needed.

However, in all these cases it is important to make sure that the sensitive data stored in IndexedDB is encrypted to ensure user privacy and security.



Cache Storage Tab

Cache Storage tab stores **data (files, API requests, images) cached by service workers** as part of providing offline and caching functionality and improving web performance for Progressive Web Apps (PWAs).



Unlike standard browser caching, **Cache storage** tab gives developers explicit control over what assets are downloaded and stored, including full HTTP headers and response bodies. You can click on each cached entry and review headers as well as preview if applicable (e.g., in case of images) or delete data.

Application Panel: Background Services Section

- **Back/Forward Cache:** Allows you to understand if **back/forward cache** is enabled successfully for your application or not.
  For more details, check Back/Forward Cache section below.
- **Speculative Loads:** Lets you debug speculative loads set with **Speculation Rules API**. It shows the speculative loading status, rule sets and speculative loading attempts. The Speculation Rules API is a new and experimental feature that can significantly improve your application's performance and user experience.
  For more details, check Speculative Loading (Experimental) section below.
- **Push Messaging and Notifications:** The **Push messaging** tab lets you record push messages for the current page's domain for up to three days and logs them.
  In the same way, the **Notifications** tab lets you record notifications received in the browser for the current page's domain for up to three days and logs them.
  For more details, check Push Messaging and Notifications section below.

Application Panel: Back/Forward Cache

What is Back/Forward Cache (bfcache)

Back/forward cache or **bfcache** is a technique that browsers are using to cache your web page entirely for a short period of time and serve it from bfcache next time you visit it (go back to it). This improves speed and user experience of your application. When you navigate away from the page, the browser keeps the old page around in the background for a short period so if you need to go back, it's available instantly.
In the Chrome browser, bfcache is enabled by default, however web applications can use certain features or APIs that prevent it from being used, such as:

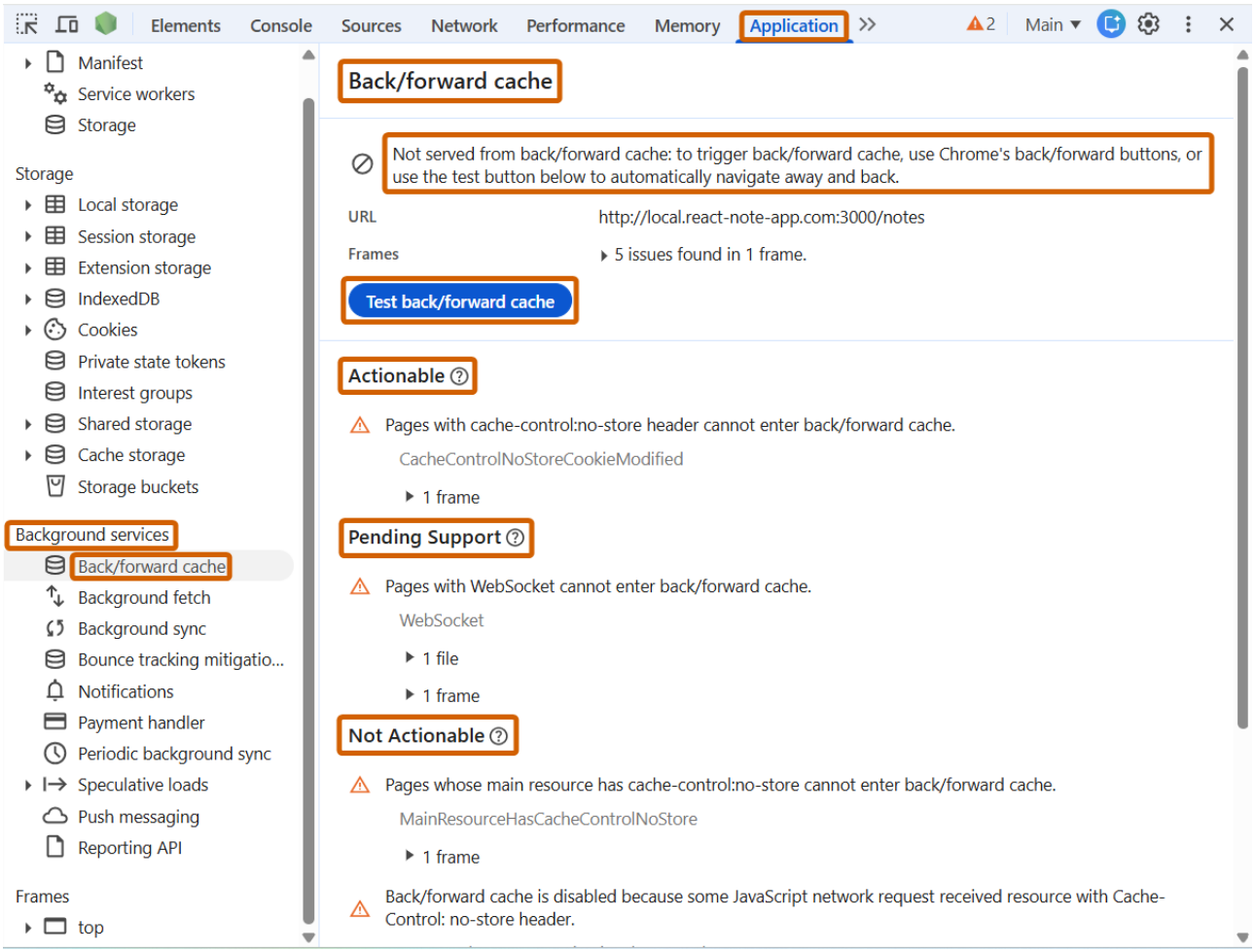
- `Cache-Control: no-store` **header:** If a page is using this header, it is a strong directive to the browser not to store any cache including bfcache. This option should be used for pages that contain personal or private information. For pages that you just want to be reasonably fresh, use `Cache-Control: no-cache`, or even better: a short cache time like `max-age=60`. Both options still allow bfcache to be used.
- **Unload handlers:** It is suggested not to use unload handlers as they are going to be deprecated because they cause a lot of issues to the application performance including inability to use bfcache. In order to prevent your application code and third party tools from using this event listener, you can set `Permissions-Policy: unload=()` which is an HTTP response header sent with the document response.
  After adding the header, if you write `window.addEventListener('unload', () => console.log('something'))` in your Console, you will see that it's not working.

- **WebSockets:** WebSockets may change data of your application in real time, that's why the browser decides that it is not accurate to use bfcache for this kind of application.
  In local development when working with frameworks like Angular or React, you may notice that bfcache is not working due to active WebSockets. For local development, if WebSockets are not a part of your application but are instead injected by the development server, you can run a production build locally to verify if bfcache is working.

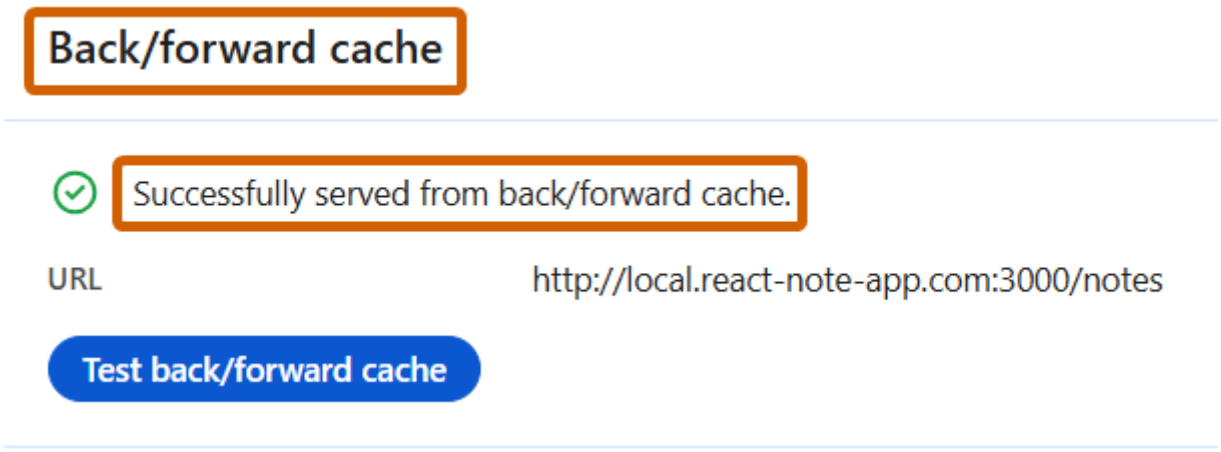
How to Debug Back/Forward Cache

To check if `bfcache` is enabled for your application, navigate to the **Application > Background services > Back/forward cache** tab and click the **Test back/forward cache** button. You will see a little flash as Chrome navigates away to a terms of service page and then back.

- If the bfcache was not enabled, you will see a message that starts like this: **`Not served from back/forward cache: to trigger back/forward cache,...`** and you will also see reasons why it was not enabled.



- If bfcache was enabled, you will see the message **`Successfully served from back/forward cache.`**



Application Panel: Speculative Loading (Experimental)

What is Speculative Loading

Speculative loading refers to the practice of performing navigation actions (such as DNS fetching, fetching resources or rendering documents) before the associated pages are actually visited, based on predictions of which pages the user is most likely to visit next.
For example, if you know that users who visit the Market Review page in a Trading application are likely to navigate to the Trading page, you can prerender that page either right after visiting the Market Review page or when hovering over the Trading page link.

The **Speculation Rules API** is an experimental technology that allows users to benefit from a performance boost by either prefetching or prerendering future navigations to give quicker or even instant page loads. It provides an alternative to the widely available `<link rel="prefetch">` feature and is designed to replace the Chrome-only deprecated `<link rel="prerender">` feature.

- **Prefetching** involves fetching some or all of the resources required to render a document (or part of a document) before they are needed, so that when the time comes to render it, this can be achieved much more quickly.
- **Prerendering** actually renders the content ready to be shown when required. Depending on how this is done, this can result in an instant navigation from the old page to the new page.

Speculation rules can be specified inside inline `<script type="speculationrules">` elements and external text files referenced by the `Speculation-Rules` response header. The rules are specified as a `JSON` structure. It is also possible to include multiple speculation rules per page.

An **eagerness setting** is used to indicate when the speculations should fire, which is particularly useful for document rules.
Below are the two useful options:

- **moderate:** On desktop, this performs speculations on hover when you hold the pointer over a link for about 200 milliseconds (or on the pointerdown event if that happens sooner). On mobile, where hover does not exist, speculations are triggered based on tap intent signals (such as pointerdown), allowing the browser to speculate only when a navigation is likely.

- **immediate:** This is used to speculate as soon as possible, that is, as soon as the speculation rules are observed.

Speculative Loading: Example

For example, let's prerender or prefetch some links for the financial application's Market Data page.

- After exploring the **Market Data** page, users are likely to click on the **Trading** page to start trading, so let's **prerender** it immediately.
- Normally, users don't view all market data details, so let's **prefetch the first visible market data detail pages** and **prerender the specific details page** that the user hovers over.
- Finally, users are likely to check their **Portfolio** page when exploring market data to see their own asset holdings, so let's **prefetch** that link.

1. Add this script in the head section of your HTML document:

```
<script type="speculationrules">
{
  prerender: [
    {
      where: { selector_matches: '.prerender' },
      eagerness: 'immediate' // On page load
    },
    {
      where: { selector_matches: '.prerender-hover' },
      eagerness: 'moderate' // On hover
    }
  ],
  prefetch: [{
    where: { selector_matches: '.prefetch' },
    eagerness: 'immediate'
  }]
}
</script>
```

2. Then add class `prerender`, `prerender-hover` or `prefetch` to your link elements (like <a> tags or framework-specific link tags):

```
<a href="https://trading.domain.com/start" class="prerender">Trading</a>
<a href="https://portfolio.domain.com/details" class="prefetch">Portfolio</a>

...

<a href="https://market-data.domain.com/details/10" class="prefetch prerender-
hover">View Details</a>
```

```
<a href="https://market-data.domain.com/details/11" class="prefetch prerender-hover">View Details</a>
```

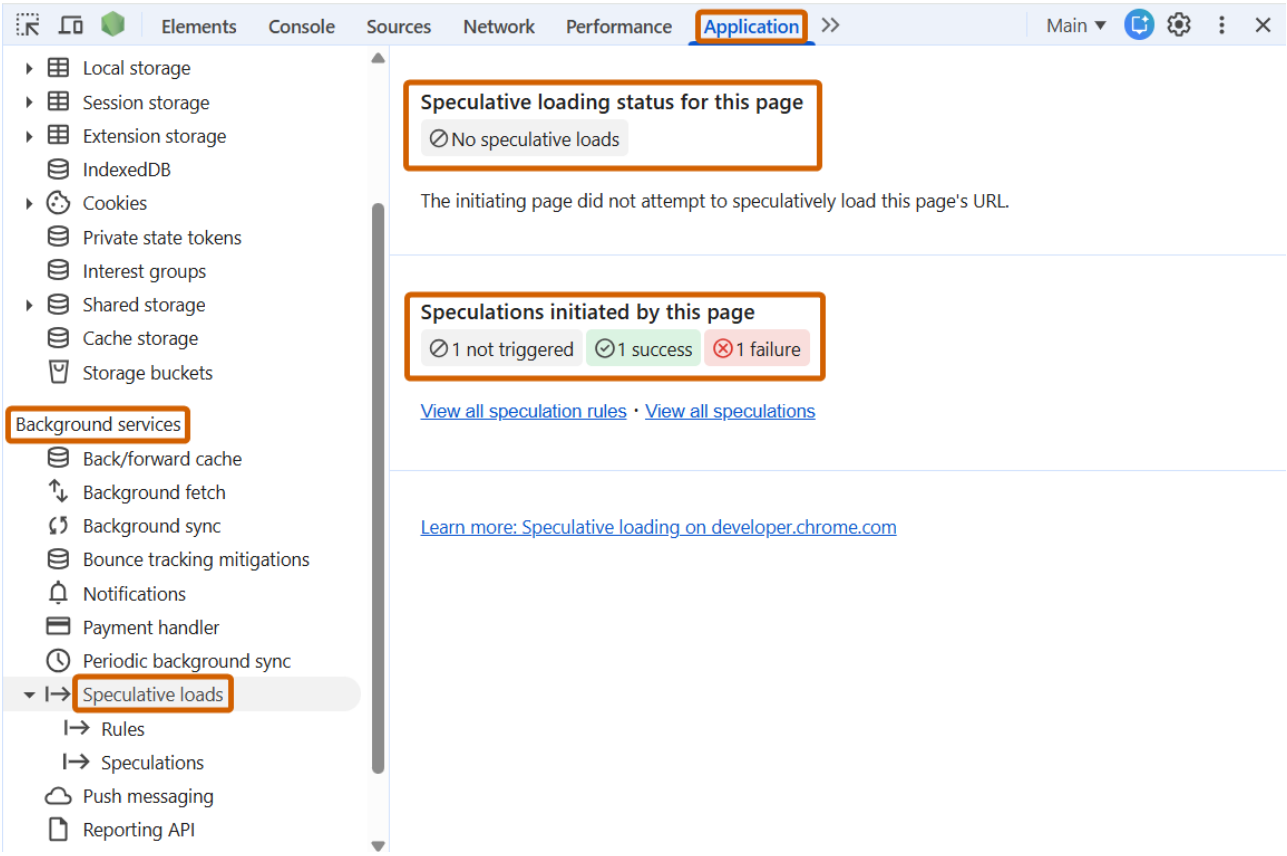
How to Debug Speculative Loading

If speculative loading rules are added to your page either via a script tag or through headers, you can click on **Application > Speculative loads** tab to inspect the rules and loaded/failed speculations initiated by the page. It lists the following three sections:

- **Speculative loads**

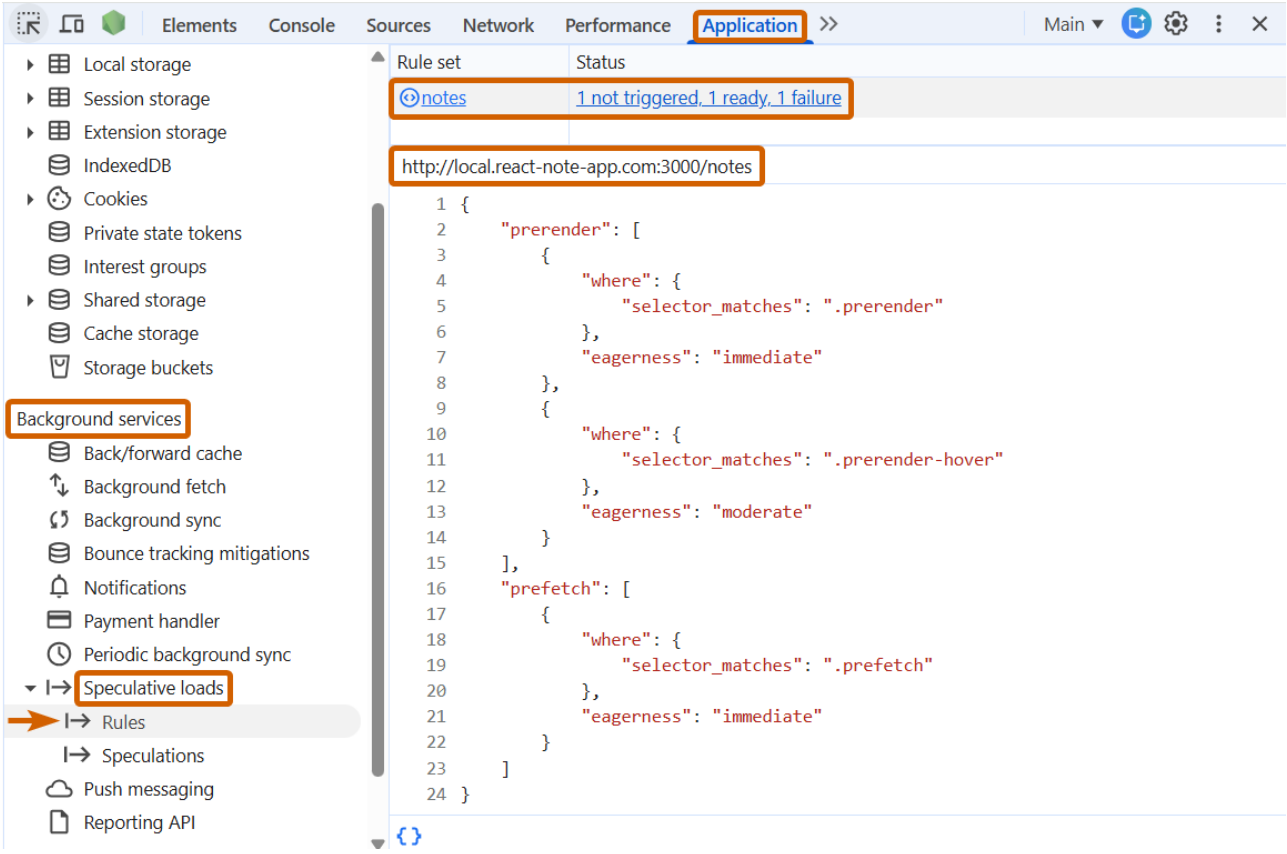
  Load the page which lists speculation rules (e.g., via a script tag) with the Application panel closed, and then after the page is loaded open the **Speculative loads** tab: You will see:

  - Summary of speculative loading of the page.
  - Speculations initiated by that page.

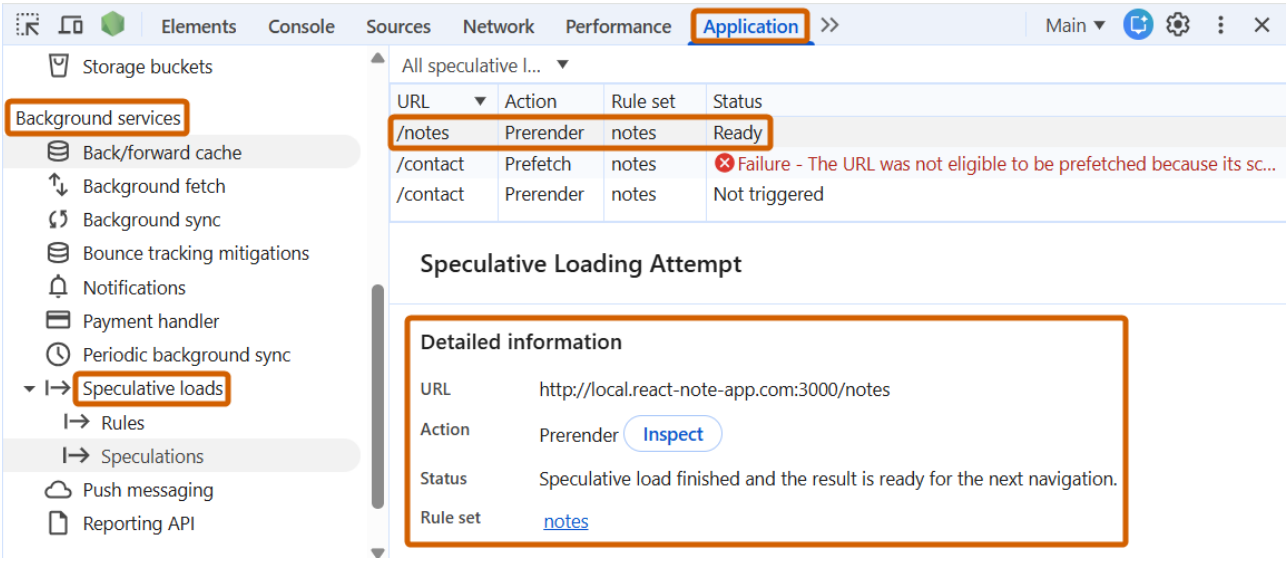


- **Speculative loads: Rules**

  - View all speculation rules specified for the page.
  - Click on each rule to view the details.



- **Speculative loads: Speculations**
  - View all speculations with their action and status:
    - **Failure** (attempted and failed)
    - **Ready** (attempted and successful)
    - **Not triggered** (specified but not triggered yet, for example, hover speculations before hovering on a specific link).
  - Click on each speculation to view the details.



Speculative Loading: Useful Tips

- The Speculation Rules API is an **experimental technology**. Please check the browser compatibility table carefully before using this in production.
- The Speculation Rules API is designed for **navigations (prefetching/prerendering entire pages)**, not for fetching individual assets. For prefetching individual assets (e.g., JS, CSS or images), you still need to use `<link rel="prefetch">`.
- At the time of writing this handbook, cross-origin URLs were only supported for same-site domains (e.g., `trading.domain.com` and `portfolio.domain.com`). This means you cannot prefetch external links like YouTube, Help pages, vendor pages, etc. from your web page. In the future, cross-site prefetches will be provided via the `Supports-Loading-Mode` header as part of the roadmap.
- Each prefetch and prerender takes resources from the browser, so it is important to **make sure adding speculation rules brings real value and performance optimizations** before adding them.
    - As mentioned earlier, speculative loading **makes sense for multi-page applications (MPAs)** rather than single-page applications (SPAs).
    - To identify which URLs are likely to be clicked and define effective speculation rules, you can enable a marketing analytics tool (e.g., Google Tag Manager) in your application to track where users interact most.

Application Panel: Push Messaging and Notifications

Push messages and **Notifications** are two separate but complementary technologies.

- **Push** is the technology for sending messages from your server to users even when they're not actively using your website.
- **Notifications** are the technology for displaying the pushed information on the user's device. It's possible to use notifications without push messaging. Also, push events may occur, but notifications might not appear if the browser or OS is blocking them.
- **Push messages** enable you to bring information to the attention of your users even when they're not using your web application.
- **Notifications** present small chunks of information to a user. Web applications can use notifications to tell users about important, time-sensitive events or actions the user needs to take.

For example, in a Notes PWA app, when a user sets notes with due dates and reminders, the server can send push messages through the service worker (if back-end synchronization is enabled) even if the user is not using the application at the moment, and the service worker will display them through the Notification API.

The real power of notifications comes from the **combination of service workers and push technology**:

- **Service workers** can run in the background and display notifications even when your application isn't visible on screen.
- **Push technology** lets you configure your server to send notifications when it makes sense for your application. A push service creates unique URLs for each subscribed service worker. Sending

messages to a service worker's URL raises events on that service worker, prompting it to display a notification.
- Your web application asks for permission to show notifications from the user, and if permission is granted, the service worker can display received messages through the Notification API.

Debugging Push Messaging and Notifications

You can debug Push messaging and notifications from the **Application > Service workers** tab. Here you can use the **Push** button to send an example Push notification to the service worker URL from DevTools. Below are the steps to follow:

1. **Configure Service Worker**

   Configure your **service worker** by adding a small code snippet in your service worker file to **catch a push message and display a notification in the browser via Notification API**:

```
self.addEventListener('push', event => {
    // Defaults
    let title = 'Push Message: Title';
    let options = {
        body: 'Push Message: Default body text'
    };

    // Guard: Check if notification permission was granted
    // In real scenarios you may need to ask the user for permission
    // and wait for permission to be given.
    if (Notification.permission !== 'granted') {
        console.warn('[Service Worker] Notification permission not granted');
        return;
    }

    // Guard: Payload may be missing
    if (event.data) {
        try {
            options.body = event.data.text();
        } catch (e) {
            console.error('Error reading push data:', e);
        }
    }

    // Display notification
    event.waitUntil(
        self.registration.showNotification(title, options)
    );
});
```

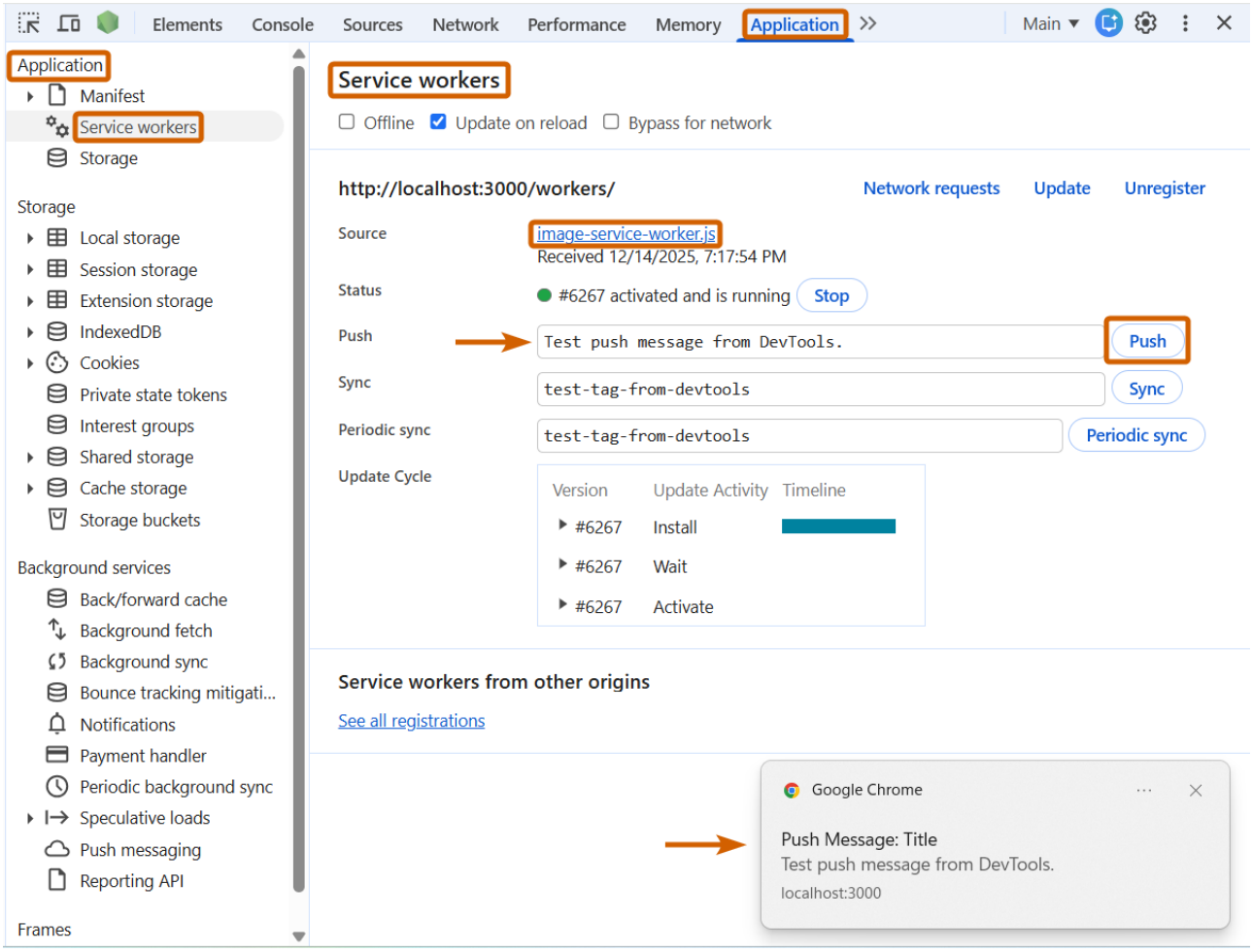
2. **Fix Permission Issues**

There are some situations where your push notification may not be displayed in the browser:

- **Browser permission is required:**
  - Your code should be configured to request permission via the browser prompt. Once permission is granted, your application can display notifications.
  - You can enable permission manually in the Chrome browser. To do this, click on the icon next to the page URL, go to **Site Settings > Notifications** and select the **Allow** option from the dropdown.
- **Your OS (Operating System) may block push notifications:** This mainly happens when the OS sets **Do not Disturb** or **Focus** modes on your PC. In these cases, check the PC **Settings**, make sure the **Do not disturb** mode is off, and ensure that it is not scheduled to turn on automatically. Restart your PC if needed for changes to take effect.

3. **Trigger a Push Message**

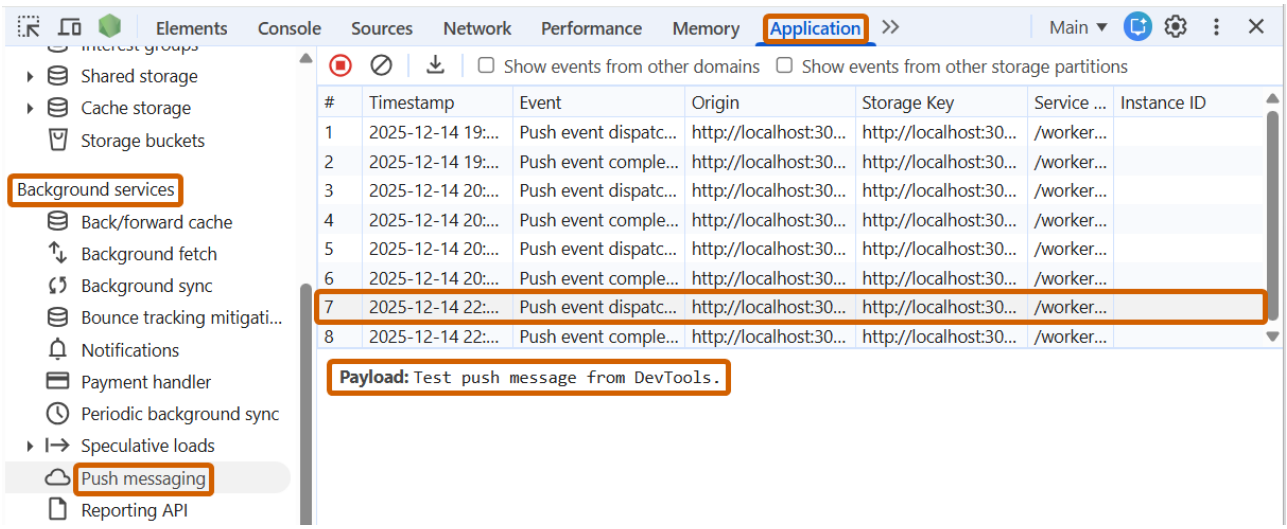
Navigate to **Application > Service workers**, write a dummy message in **Push** input and press the **Push** button to push a message from DevTools to your service worker. You will see it appearing in the browser (it may be at the top, at the bottom, with different UI based on Chrome settings and OS).



> **Note:** Before pushing messages, make sure to turn on the red record icon (⦿) in **Background services > Push messaging** and **Background services > Notifications** sections to start recording push messages and notifications for up to 3 days after enabling it.

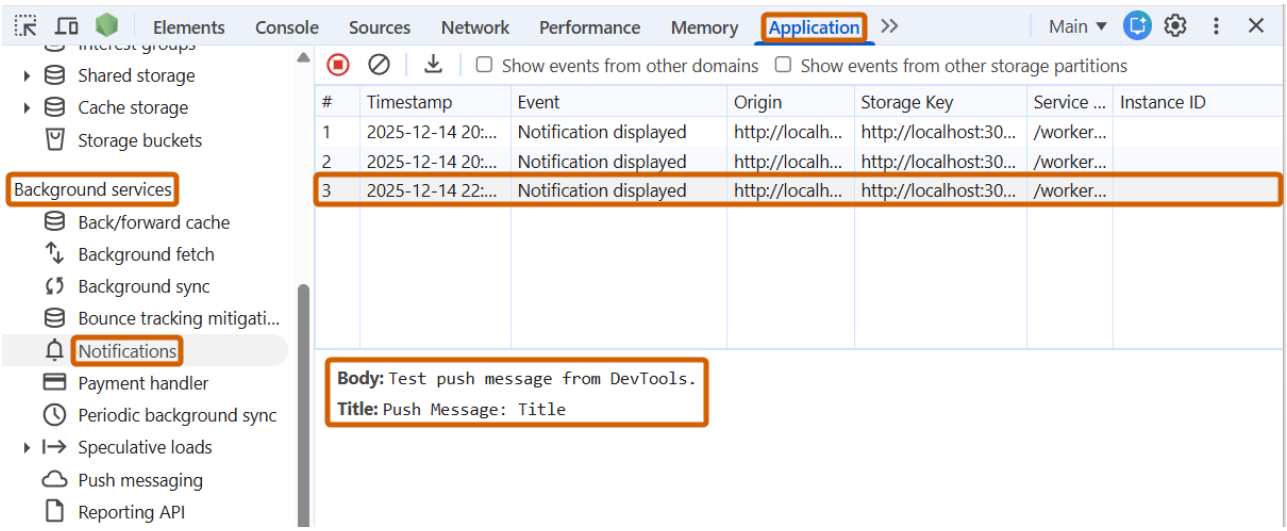
4. **View Pushed Messages**

   Navigate to the **Background services > Push messaging** tab to view pushed messages.



5. **View Displayed Notifications**

   Navigate to the **Background services > Notifications** tab to view displayed notifications.



Application Panel: Common Use Cases

- **Inspecting and Debugging Cookies**: Verify that authentication-related cookies are being set correctly. You can inspect their domain, path, expiration date and security attributes (e.g., `HttpOnly`, `Secure` and `SameSite`).

- **Managing Client-side Storage**: View, edit and delete data stored on the client side to debug application state. This includes **Local Storage**, **Session Storage** and **IndexedDB**.
- **Resetting Application State**: Use the **Clear site data** button to completely clear out all storage, caches and service workers for an application. This is essential for testing the **first-load** experience for new users.
- **Debugging Service Workers**: Inspect the lifecycle of your service workers (installing, activating, running) and their cached resources for Progressive Web Apps (PWAs) and offline capabilities. Also, observe push messages sent to service workers and notifications displayed by service workers for up to three days.
- **Verifying the Web Application Manifest**: Inspect your `manifest.json` file to ensure your Progressive Web App (PWA) meets the requirements to be installable, including its icons, theme colors and start URL.
- **Debugging Back/forward Cache**: Check if the **bfcache** feature works for your web application. If not, get actionable insights on how to fix it and benefit from this great performance optimization.
- **Inspecting Speculation Rules**: Check speculation rules to see which ones are applied, which pages were **prefetched** or **prerendered** according to the rules, and which ones failed. Use actionable insights to fix failed **prefetched** or **prerendered** pages to align with speculation rules requirements and benefit from this performance optimization.

1.8 Sources Panel

The **Sources** panel can be used to view, edit and debug your website's resources, such as stylesheets, JavaScript files and images.

Sources Panel: How to Open

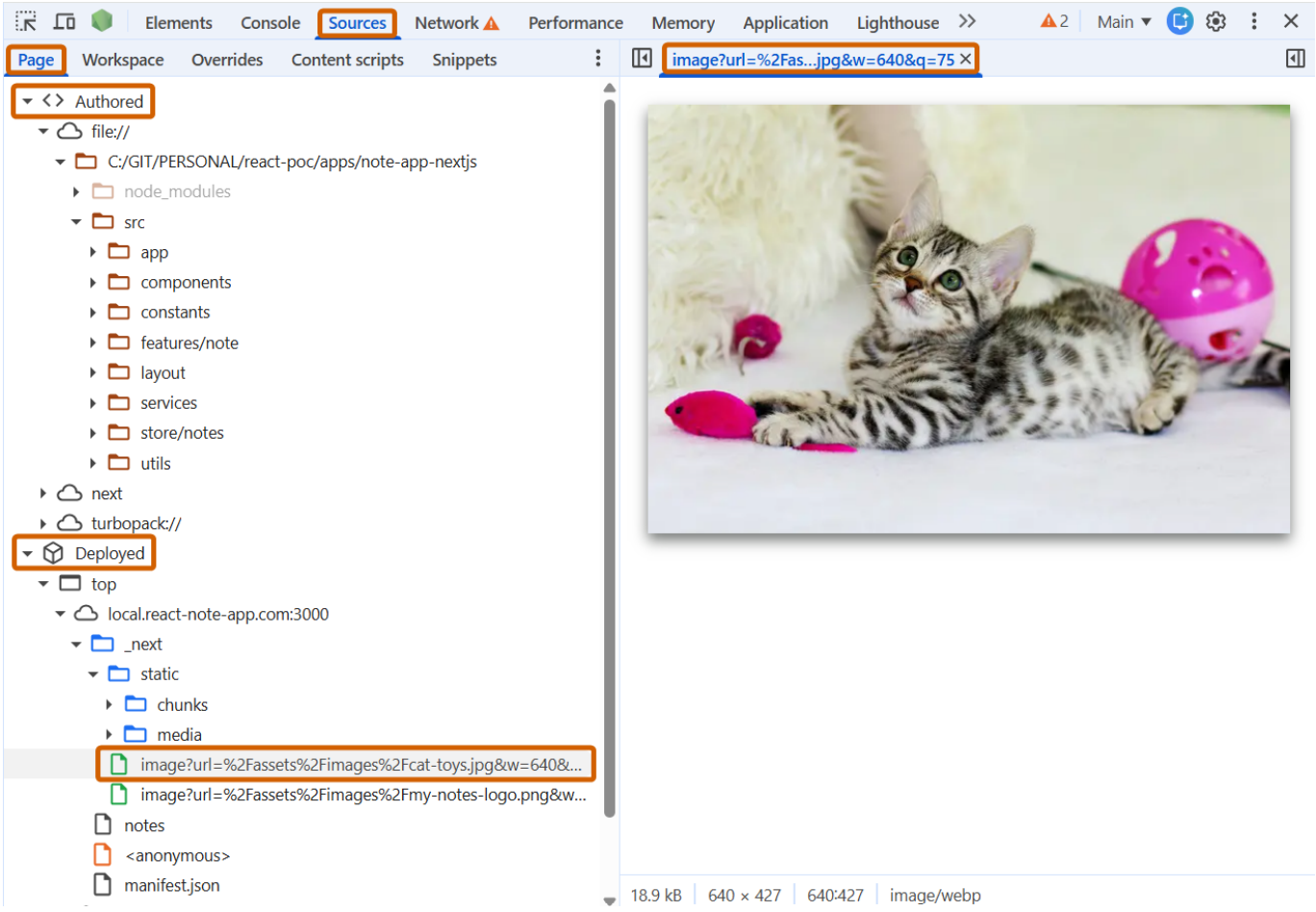
To open the **Sources** panel, open DevTools and click the **Sources** tab, or select it from the **Command Menu**.

Sources Panel: Key Features

The **Sources** panel provides several powerful features:

1. View File Hierarchy

If you click on the **Page** tab, you will be able to review your project's source code file hierarchy.



You will see two main sections:

- **Authored**: Contains original sources (current local working files in case of local development). Authored appears when source maps are available. Many web applications choose not to include full source maps or include limited source maps in production.
- **Deployed**: Contains final sources the browser sees (e.g., bundled, minified files).

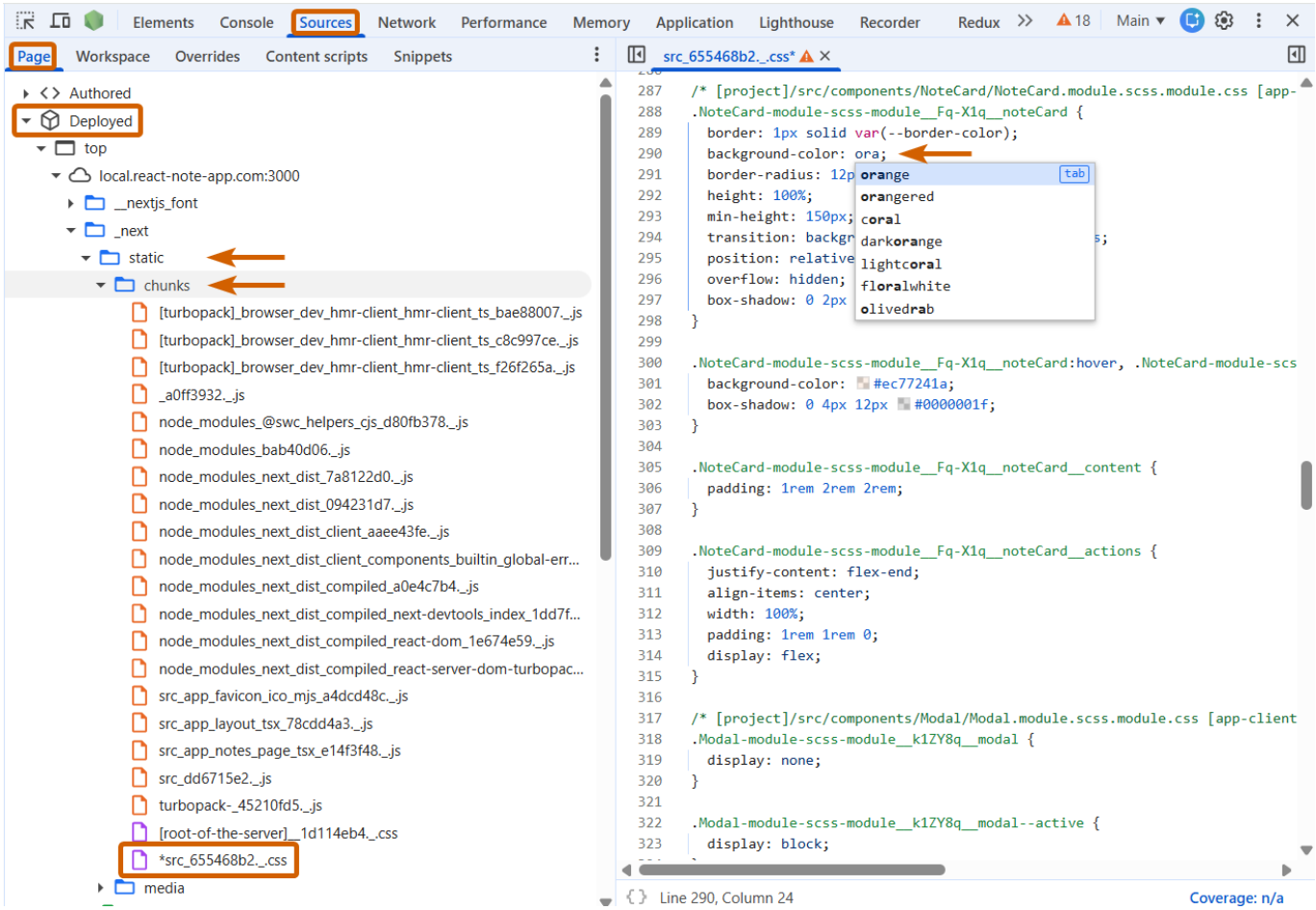
Clicking any type of file in the **Page** tab opens it in the **Editor** tab, where you can view source code or preview images directly. This is also essential for finding the right file and line to set your debugger breakpoint.

2. Edit CSS and JavaScript

If you click on a file, you can **update your JavaScript/CSS files directly in the Editor**. However, since in modern front-end frameworks JavaScript and CSS files come minified and bundled, it is hard to find the correct location in the code to edit. For local development, you can explicitly configure your build so that bundles are not minified in order to be able to make changes easily.

- For CSS, write new CSS or edit existing styles directly in the Editor and you will see the results immediately.
- For JavaScript changes to take effect, press `Ctrl+S` (Windows, Linux) or `Command+S` (macOS). DevTools doesn't re-run a script, so the only `JavaScript` changes that take effect are those that you make inside functions.

Besides providing the ability to edit CSS and JavaScript files, the **Editor** tab also underlines and shows inline error tooltips next to syntax errors and other issues, such as **failed CSS `@import` and `url()` statements** and **HTML `href` attributes with invalid URLs**.

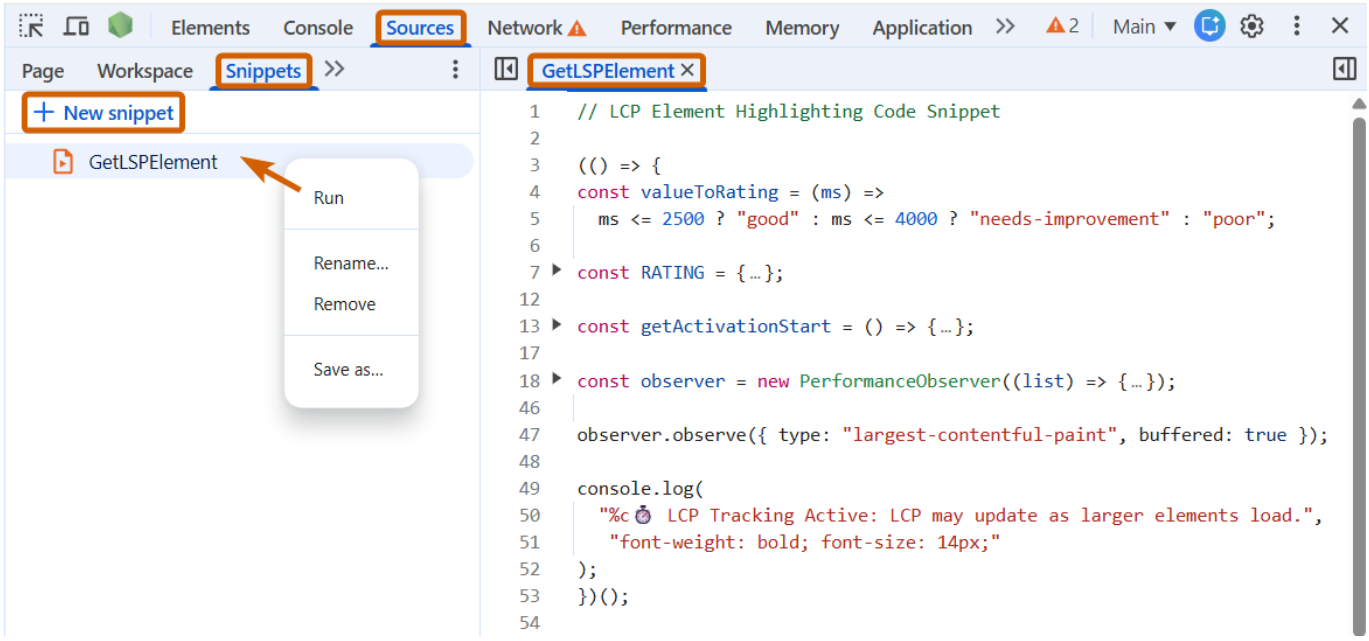


3. Create, Save and Run Snippets

Snippets are a very easy and useful, often overlooked feature of Chrome DevTools. If you find yourself running the same code in the Console repeatedly, you can save that code as a snippet instead. Snippets have **access to the page's JavaScript context**, and they serve as an alternative to **bookmarklets**.
DevTools saves each Snippet to your file system, so you don't have to rewrite the same code again and can reuse them across different applications.
Considering all of this, snippets basically act as **mini browser extensions**.

To leverage Snippets functionality, click the **Snippets** tab. Here you have the following common actions:

- **Create New**: Click **+ New Snippet** to create a new snippet.
- **Run**: Right-click the snippet and select **Run** to run it. If the snippet sets up event listeners or timers, they will be lost after a page refresh, so you need to run the snippet again.
- **Rename**: Double-click the created snippet or right-click it and select **Rename** from the context menu.
- **Remove**: Right-click the snippet and select **Remove** to delete it.
- **Save as...**: Right-click the snippet and select **Save as...** from the context menu to save the snippet as a separate JavaScript file.



Example: You can write a **performance monitoring JavaScript** snippet which detects and highlights the LCP candidate elements while browsing the website using `PerformanceObserver` API.
To view this example in action, check the PerformanceObserver Interface: Example section.

4. Debug JavaScript

Using the **Page** tab, you can locate the exact source file and the specific line of code you want to debug. From there, you can add a breakpoint and debug your code directly in the browser.
To continue with the **Debug JavaScript** feature, check the Browser Debugger chapter.

5. Overrides

The **Local Overrides** feature allows you to set up a folder on your local machine and temporarily override resources of your web application such as API responses, response headers, HTML, CSS, JS and image files to test different scenarios locally.
To continue with the **Local Overrides** feature, check the Local Overrides section.

Sources Panel: Common Use Cases

- **Review source code file hierarchy** and quickly **spot syntax errors** in the **Editor** tab.
- **Edit CSS/JavaScript files in place** to test different scenarios (more convenient for non-bundled CSS/JavaScript files).
- **Automate repeated tasks** (e.g., performance monitoring tasks) using **Snippets**.
- **Locate the necessary file and add breakpoints** to debug JavaScript logic.
- **Override web application resources** (API response, response headers, JS, CSS, HTML and images) using a local folder to debug issues locally before pushing to production.

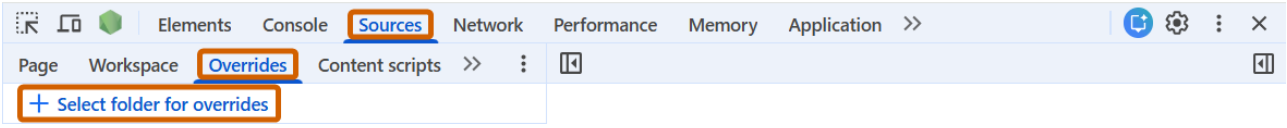
1.9 Local Overrides

The **Local Overrides** feature allows you to set up a folder on your local machine and temporarily override resources of your web application such as API responses, response headers, HTML, CSS, JS and image files to test different scenarios locally.

Local Overrides: How to Open and Setup

To set up the **Local Overrides** feature, follow these steps:

- Open DevTools and click the **Sources > Overrides** tab, or run the **Show Overrides** command from the Command Menu.
- Click on **Select folder for overrides** and choose a folder from your local machine where overrides will be saved.



After this quick setup, you can start using the **Local Overrides** feature.

Local Overrides: Key Features

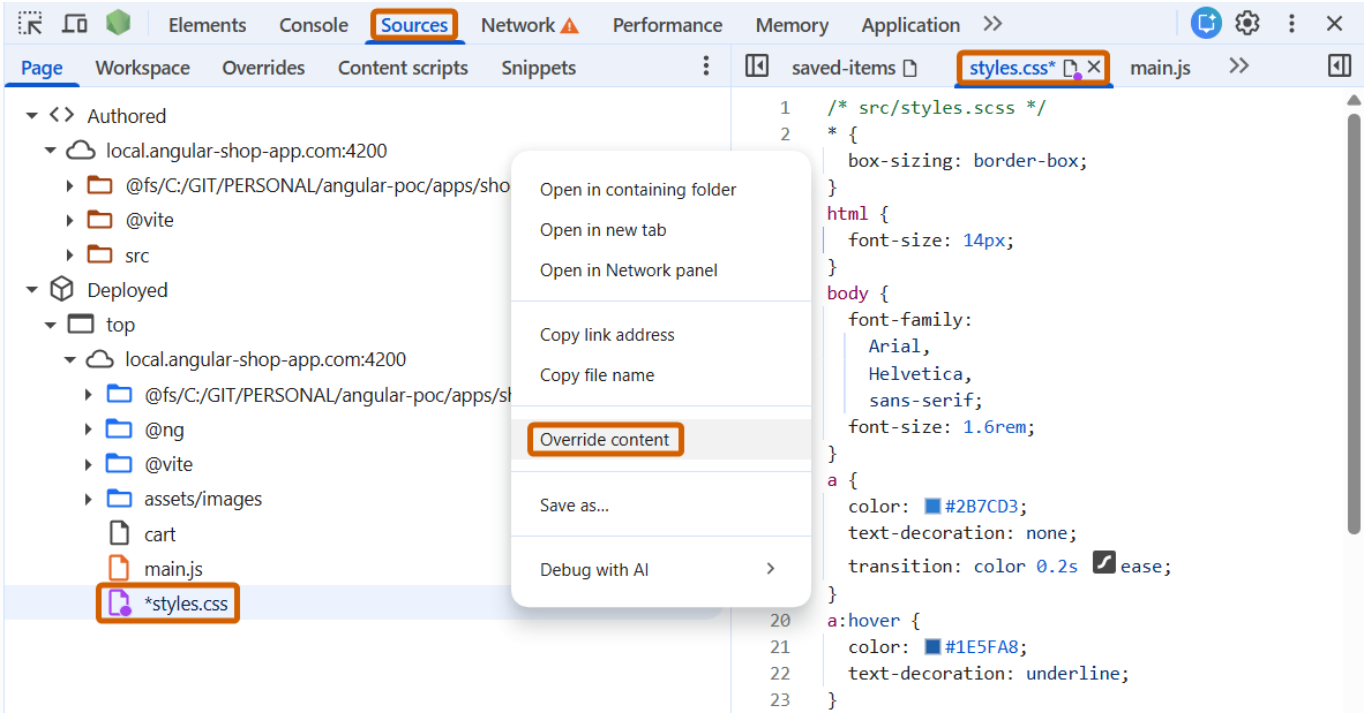
Below are key features that you will benefit from:

Override Web Content

To override web content, perform the following steps:

- Go to the **Page** tab in the **Sources** panel, right-click any source (e.g., CSS, JS, HTML) and choose **Override content** from the opened context menu. The content of the file will be displayed in the **Editor** tab, and you can start editing.
- After changes are made, `*` will be shown next to the file name indicating changes are not saved automatically. You need to press `Ctrl+S` for Windows / Linux and `Command+S` for macOS to save the changes.

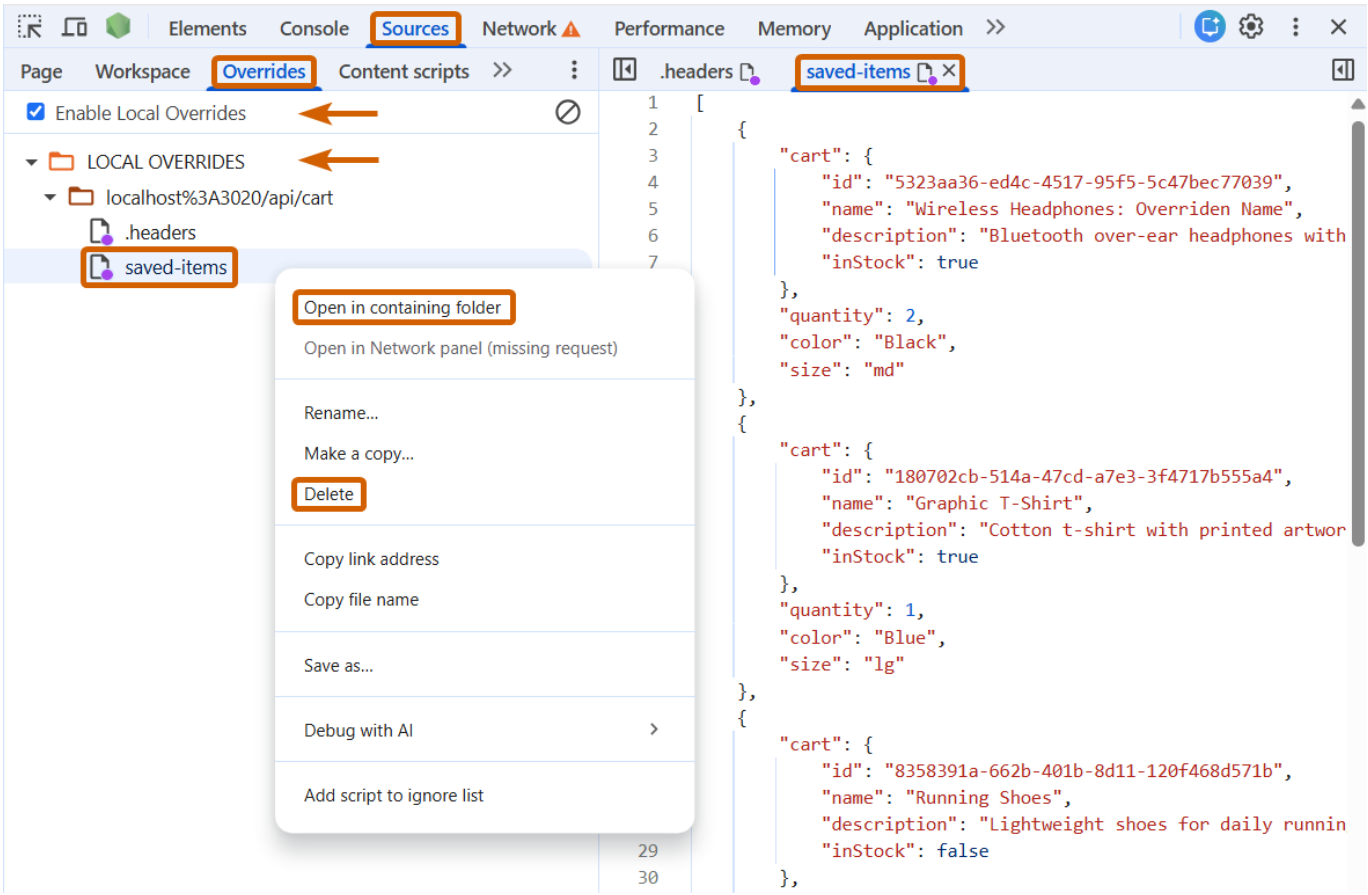
> **Note:** For modern frameworks, you can override only files that are listed under the **Deployed folder**. In most cases, since the file is bundled and minified, it is hard to find the location where changes should be made, so you may need a local setup to skip minification in order to use the **Local Overrides** feature effectively.



Override HTTP response

To override the HTTP response of your API, right-click the resource in the **Network** panel and choose **Override content**.

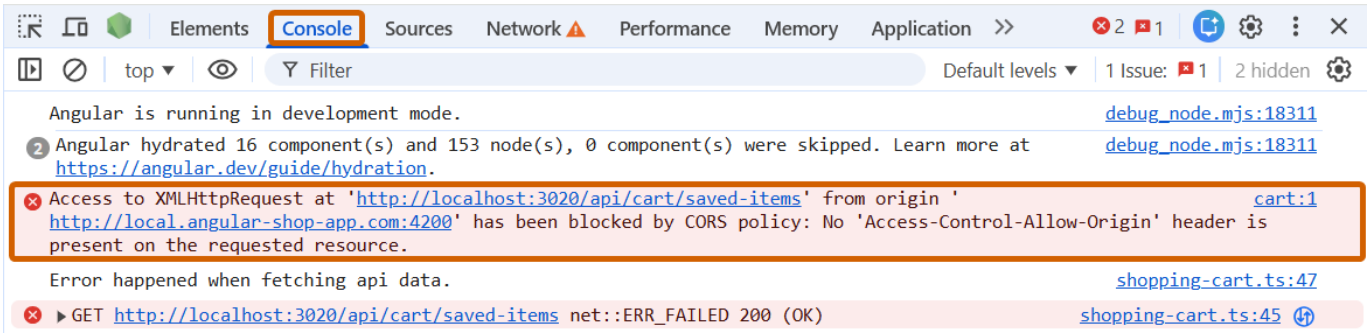
After selecting **Override content**, you will be directed to the **Sources > Overrides** tab, where you can view the existing response and start overriding it.



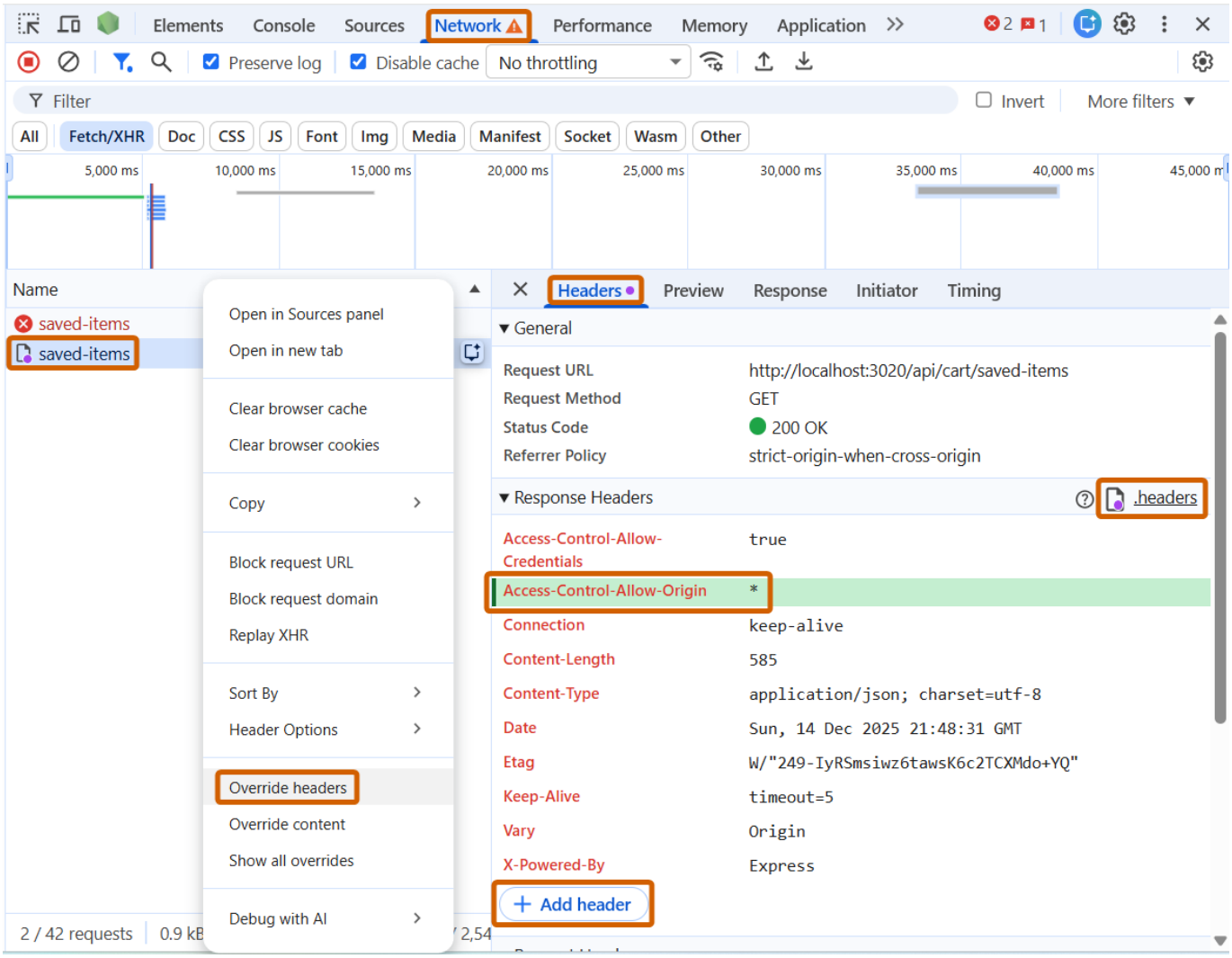
Override HTTP response headers

To override response headers for a resource, right-click the resource in the **Network** panel and choose **Override headers** from the context menu.

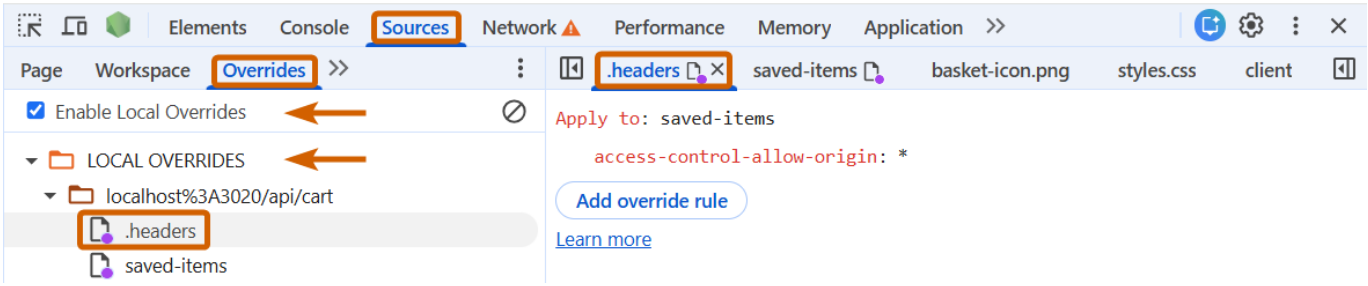
To show how useful this feature is, let's explore the following example: imagine an API request fails with CORS policy and blocks your local development process.



Instead of doing a complex setup, you can override the CORS response header locally to allow all origins by setting `Access-Control-Allow-Origin: *` and continue your local development without waiting for the back-end changes.

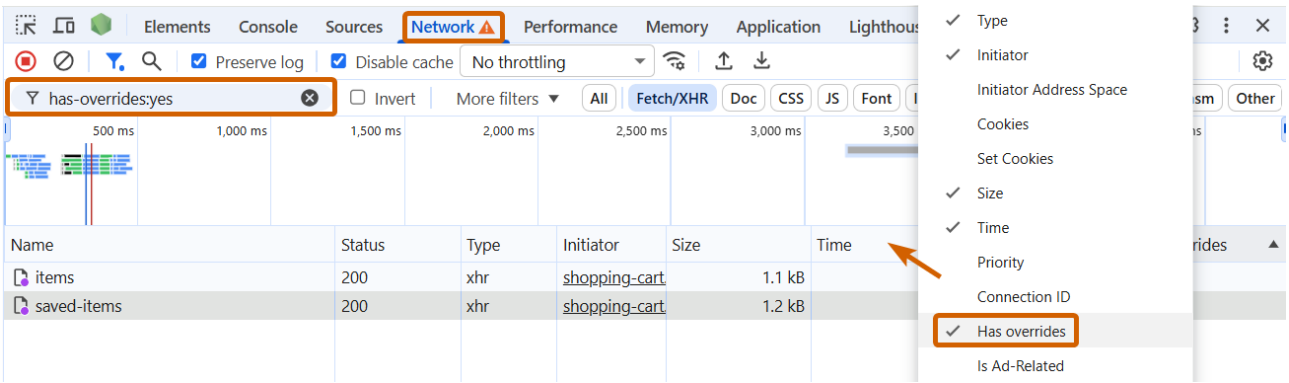


The overridden resources will be marked with a purple dot icon (●) to help you easily identify which resources are overridden and which are not.
After overriding the headers, the **.headers** link will appear in your request **Headers** tab, next to the **Response Headers** section heading. You can click it to view overridden headers in the **Sources** panel, and even make them more generic to work with all of your resources instead of a specific API URL:



Local Overrides: Managing and Filtering

- From the **Overrides** tab, you can right-click the overridden resource and use some useful context-menu options:
    - **Delete**: Remove the specific override.
    - **Open in containing folder**: Open the folder on your local machine where overrides are stored.
- To view which resources were overridden in the **Network** panel, you can right-click the **Network panel Requests table** header and choose the **Has overrides** option. After that, a new column named **Has overrides** will be added to the table of requests.
  Alternatively, you can use `has-overrides:content`, `has-overrides:headers`, `has-overrides:yes` filters in the **Network panel Filter bar** to view requests which were overridden.



Local Overrides: Common Use Cases

Local Overrides is an often overlooked feature, but it provides powerful capabilities for local debugging. The main advantage of the **Local Overrides** feature is that your changes are saved in a folder on your local machine, so they are **preserved even after refreshing the page**. In contrast, changes made directly in the **Elements** panel (e.g., CSS edits) are temporary and **will be lost after a refresh**.

Below are some use cases to accelerate your debugging flow:

- **Override broken authentication or CORS response headers** to do appropriate local testing without waiting for the back-end fixes.
- **Mock an API response** to test changes locally before pushing to production for APIs that are not available locally. Alternatively, you can mock an API response based on given UI prototypes and start working on the UI part even if the API is not ready yet.
- **Override source files (e.g., HTML, CSS, JS)** from the **Sources > Page** tab and check some performance issues (e.g., poor LCP or CLS) quickly locally before applying them to your codebase.
- **Send your local overrides to team members** from your local folder for collaboration or attach them to the bug ticket for further debugging.

> **Note:** If **Local Overrides** is enabled, DevTools will automatically disable the cache because Local Overrides doesn't work with the enabled cache.

1.10 AI Innovations Feature (Experimental)

AI Innovations is an experimental feature which integrates **Gemini AI** directly into Chrome DevTools. It allows you to streamline your debugging journey by:

- Having a conversation about the page you are inspecting.
- Understanding Console errors and warnings better with AI.
- Getting auto-generated code suggestions.
- Auto-labeling performance traces.

Gemini AI automatically has context about your page's DOM, network requests and source code, helping you understand complex issues and receive suggestions for fixes.

AI Innovations: Requirements

To use the AI assistance panel, make sure that you:

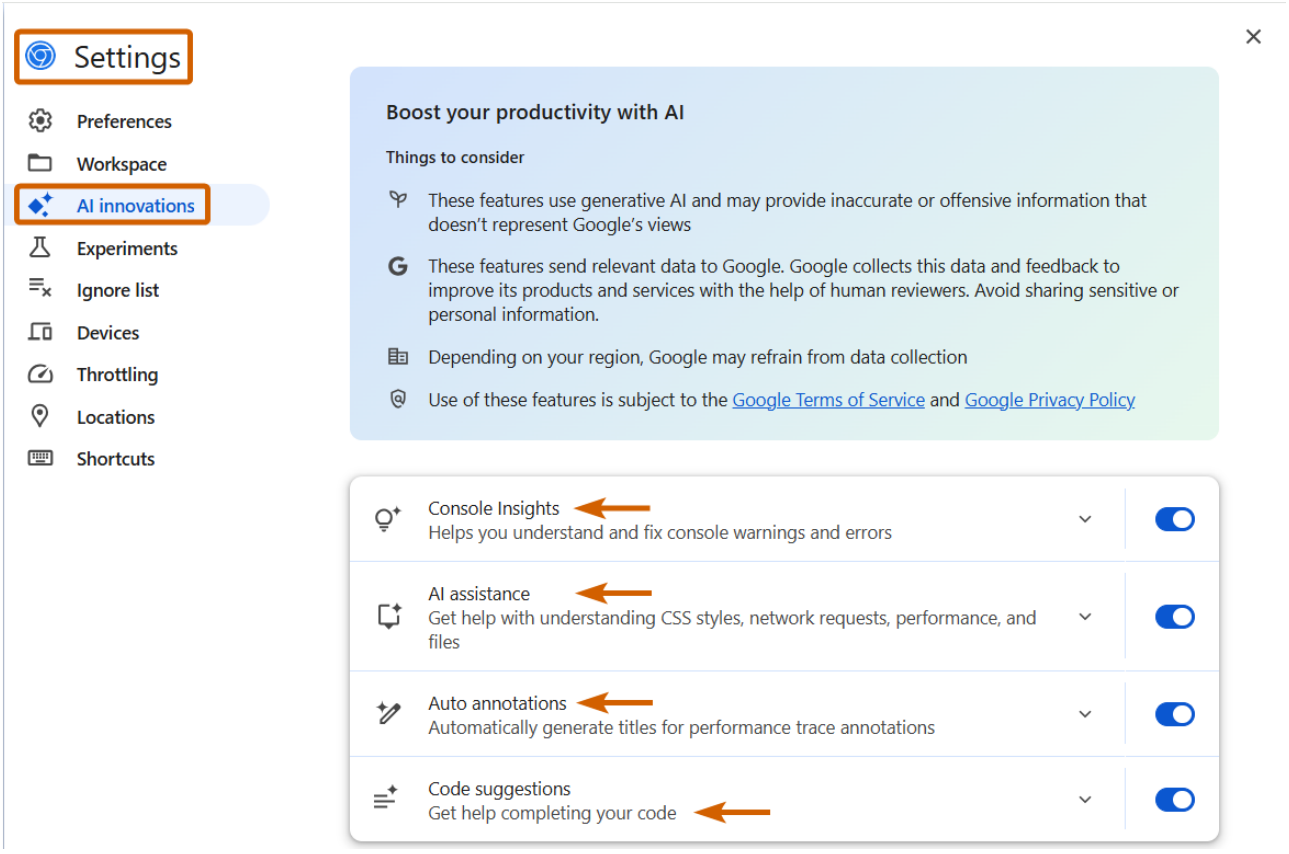
- Are **at least 18 years old** and in one of the **supported locations**.
- Are using the **latest version of Chrome**.
- Are **signed in to Chrome with your Google Account**.
- Have **English (US)** selected in **Settings > Preferences > Appearance > Language** in DevTools.
- Have enabled the setting **Settings > AI Innovations** in DevTools.

> **Note:**
>
> - Since this is an experimental feature, the best approach is to check the official requirements at the moment you read this handbook.
>   You can find them here: goo.gle/devtools-ai-reqs
> - If your Chrome installation is managed by an organization, your administrator needs to enable AI Innovations using an enterprise policy.
>   You can read more here: goo.gle/devtools-ai-enterprise

AI Innovations: How to Enable in Settings

1. Open DevTools and go to the **Settings** tab (click the ⚙ icon or press `F1`).

2. Under the **AI innovations** section, enable the features you need, such as **Console Insights**, **AI assistance**, **Auto annotations** and **Code suggestions**.



AI Innovations: AI Assistance

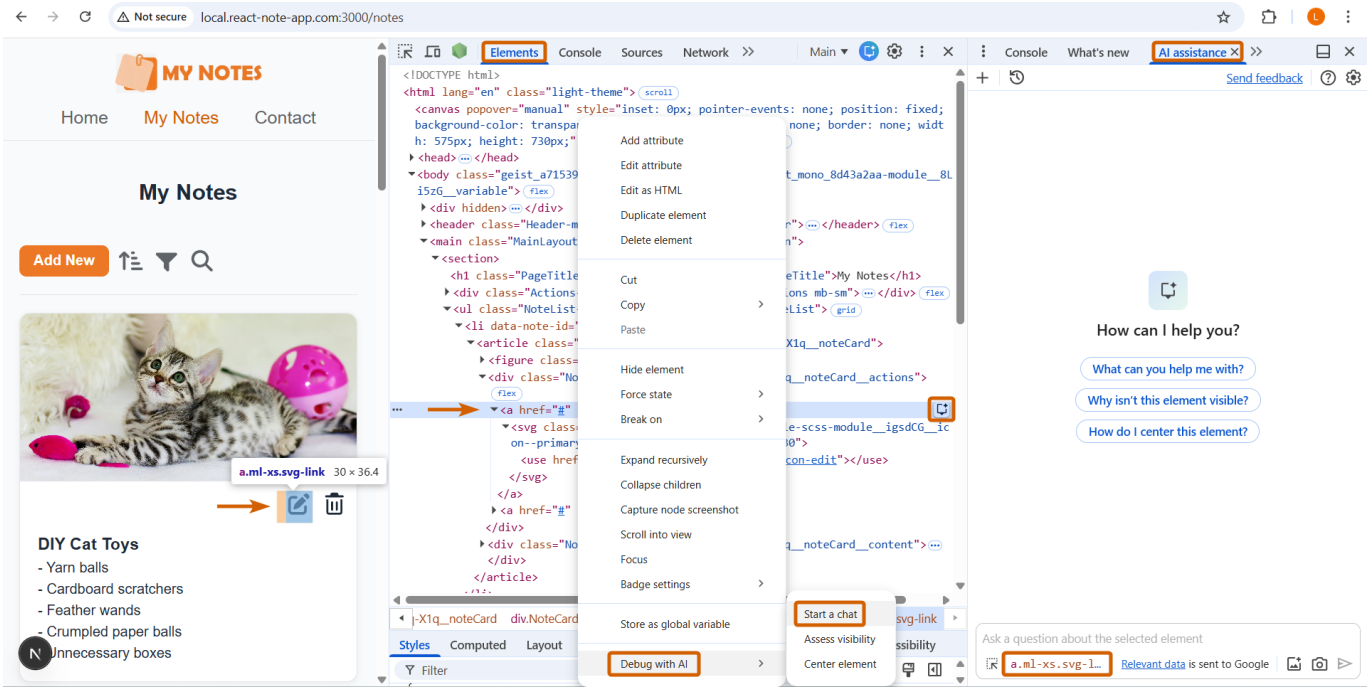
The **AI Assistance** feature allows you to have a conversation with Gemini AI directly inside DevTools about the page you are inspecting.

AI Assistance: How to Start a Conversation

The primary way to start a conversation with AI Assistance is by providing context directly from a DevTools panel:

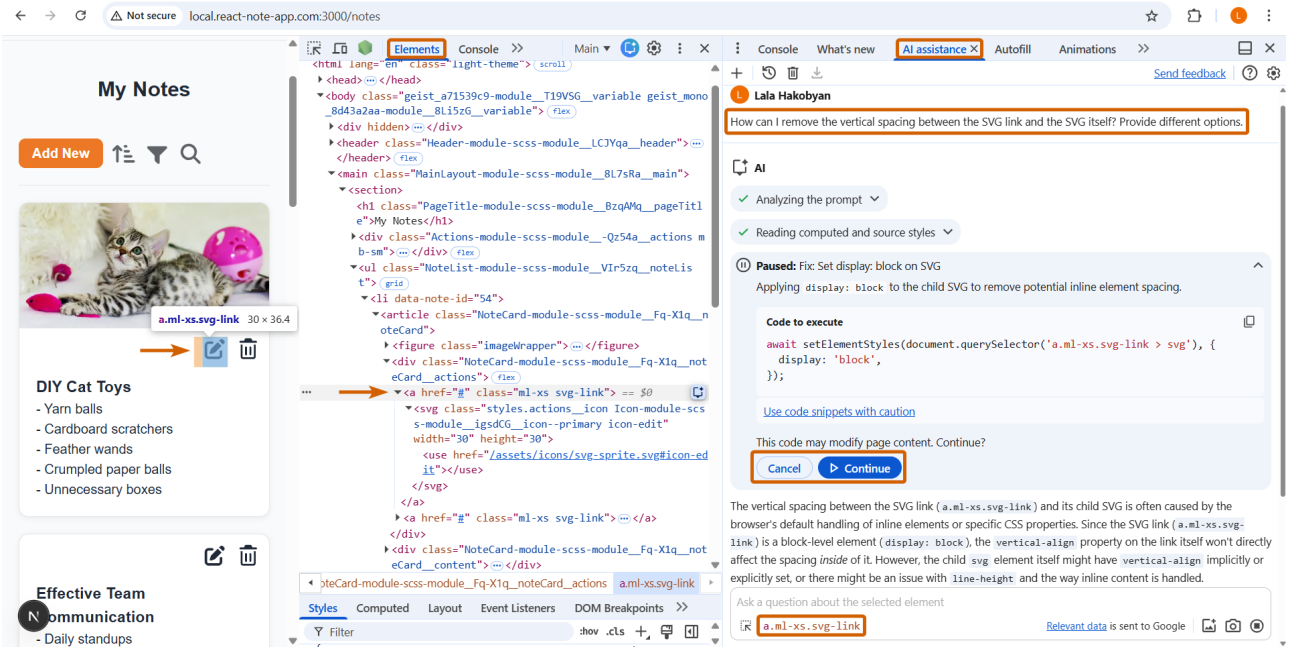
- Right-click an item you want to ask about (for example, a DOM element, a network request, or a source file name).
- Select **Debug with AI** from the context menu and choose **Start a chat** in the opened context sub-menu.
- Alternatively, hover over the item and click the small **AI Assistance** icon that appears. It will also open the **AI Assistance** panel.

After the panel opens, AI Assistance automatically selects the context and displays it in the bottom left corner of the panel. It also provides default prompts to help you get started. You can choose one of them or start a new conversation.



AI Assistance in Different Panels

- **Elements Panel:** Click any element in the Elements panel and ask the AI about style explanations and fixes.

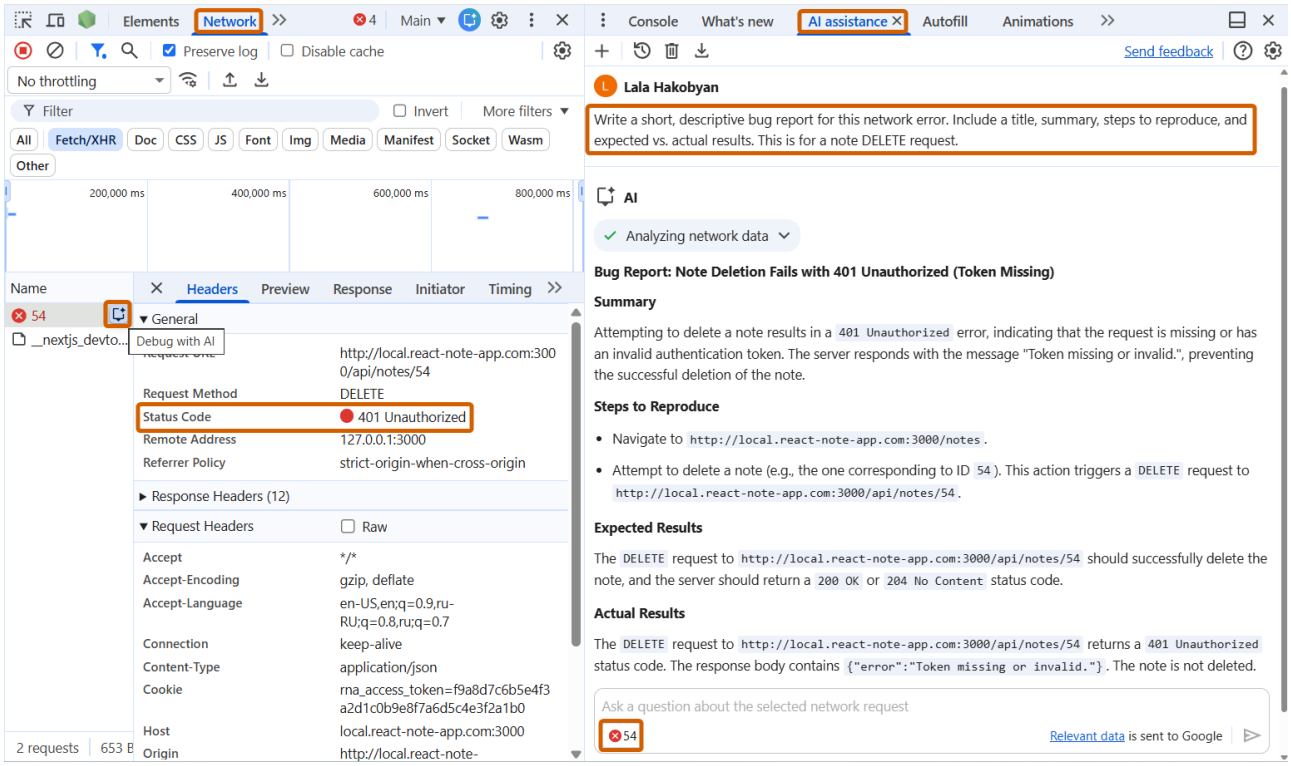


When chatting with AI Assistance or when it provides a response, it may request to execute some code snippets to get more context about the problem you are debugging.
In these cases it will provide **Continue** and **Cancel** buttons. You need to review the provided code snippet carefully and choose whether to allow it by clicking **Continue**, or cancel it by clicking **Cancel**.

Prompt Example:

```
How can I remove the vertical spacing between the SVG link and the SVG itself?
Provide different options.
```

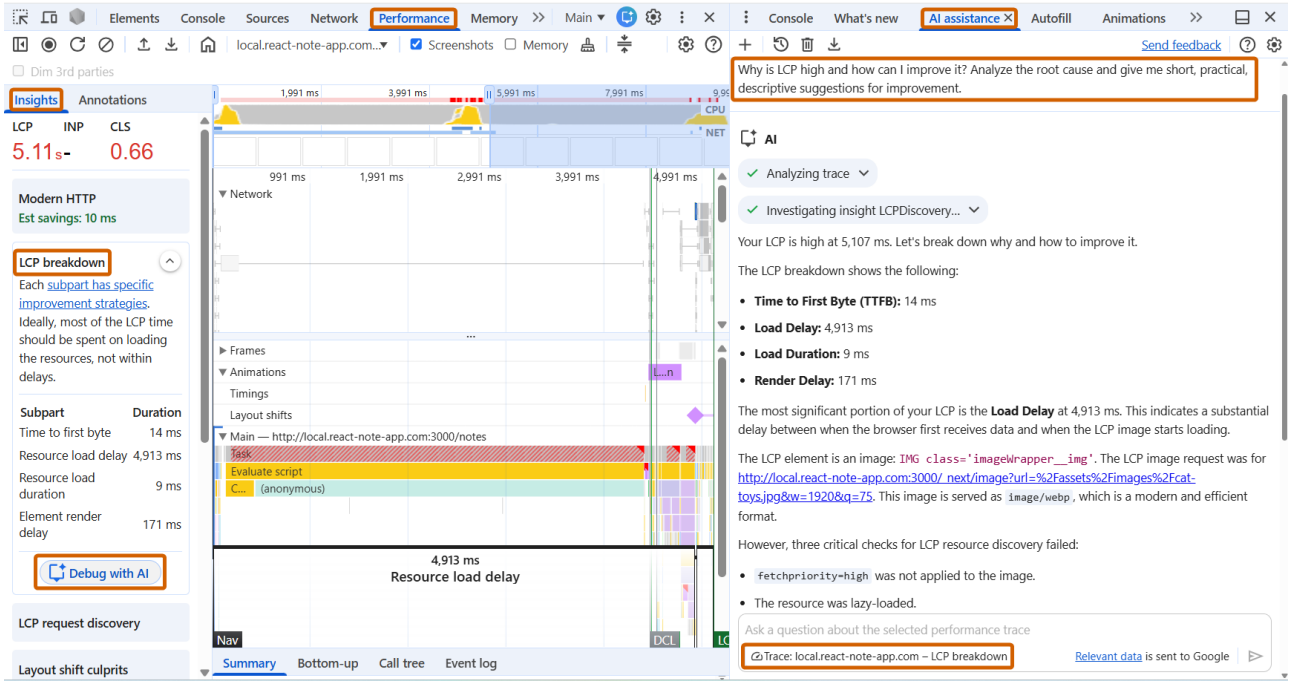
- **Network Panel:** Click a network request and ask AI to analyze it, extract headers and provide suggestions for the root cause of the issue.



Prompt Example:

```
Write a short, descriptive bug report for this network error.
Include a title, summary, steps to reproduce and expected vs. actual results.
This is for a note DELETE request.
```

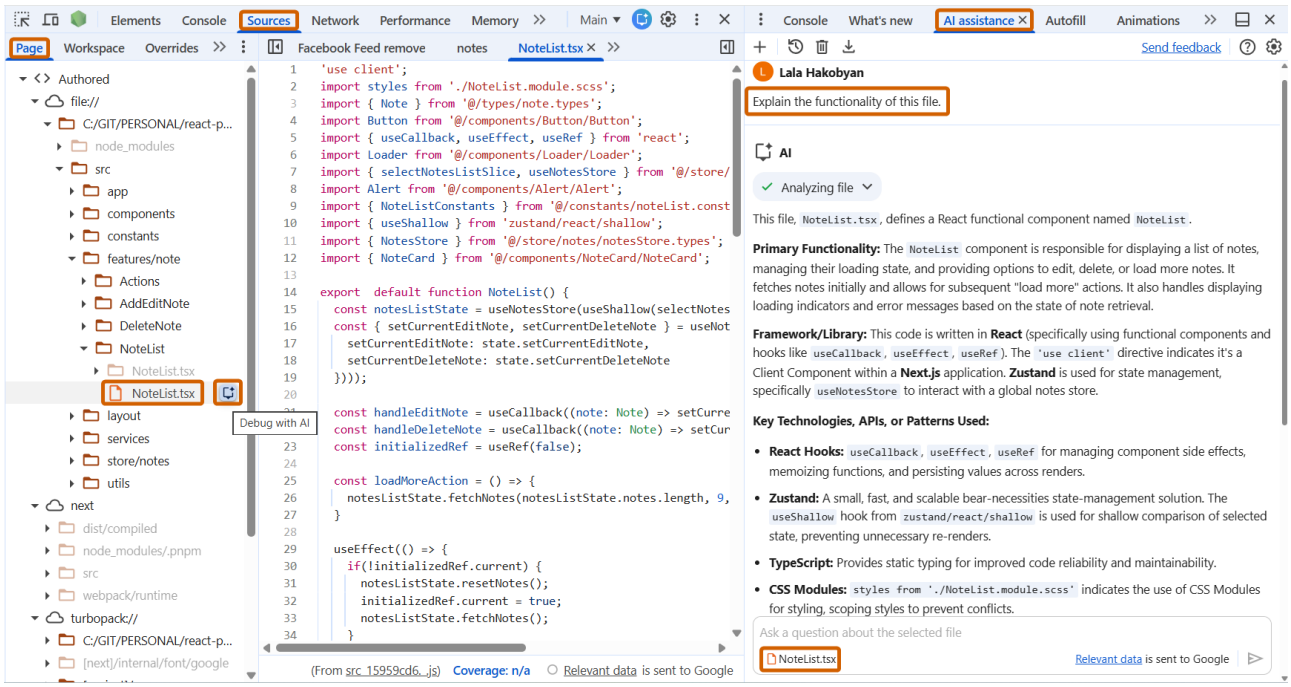
- **Performance Panel:** Expand a performance insight section in the **Performance > Insights** tab and click the **Debug with AI** button to ask AI to help you understand the cause of poor metrics and suggest fixes.



Prompt Example:

```
Why is LCP high and how can I improve it? Analyze the root cause and give me short, practical, descriptive suggestions for improvement.
```

- **Sources Panel:** Click a source file and ask AI to explain what the file does.



Prompt Example:

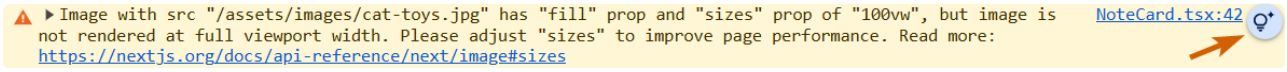
```
Explain the functionality of this file.
```

AI Innovations: Console Insights

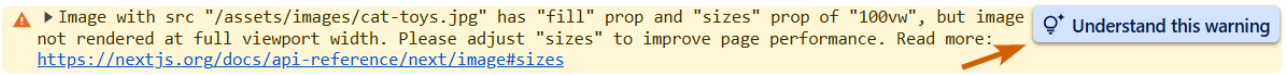
The **Console Insights** feature provides AI-generated explanations for warning and error messages in the **Console** panel.

To generate Console Insights:

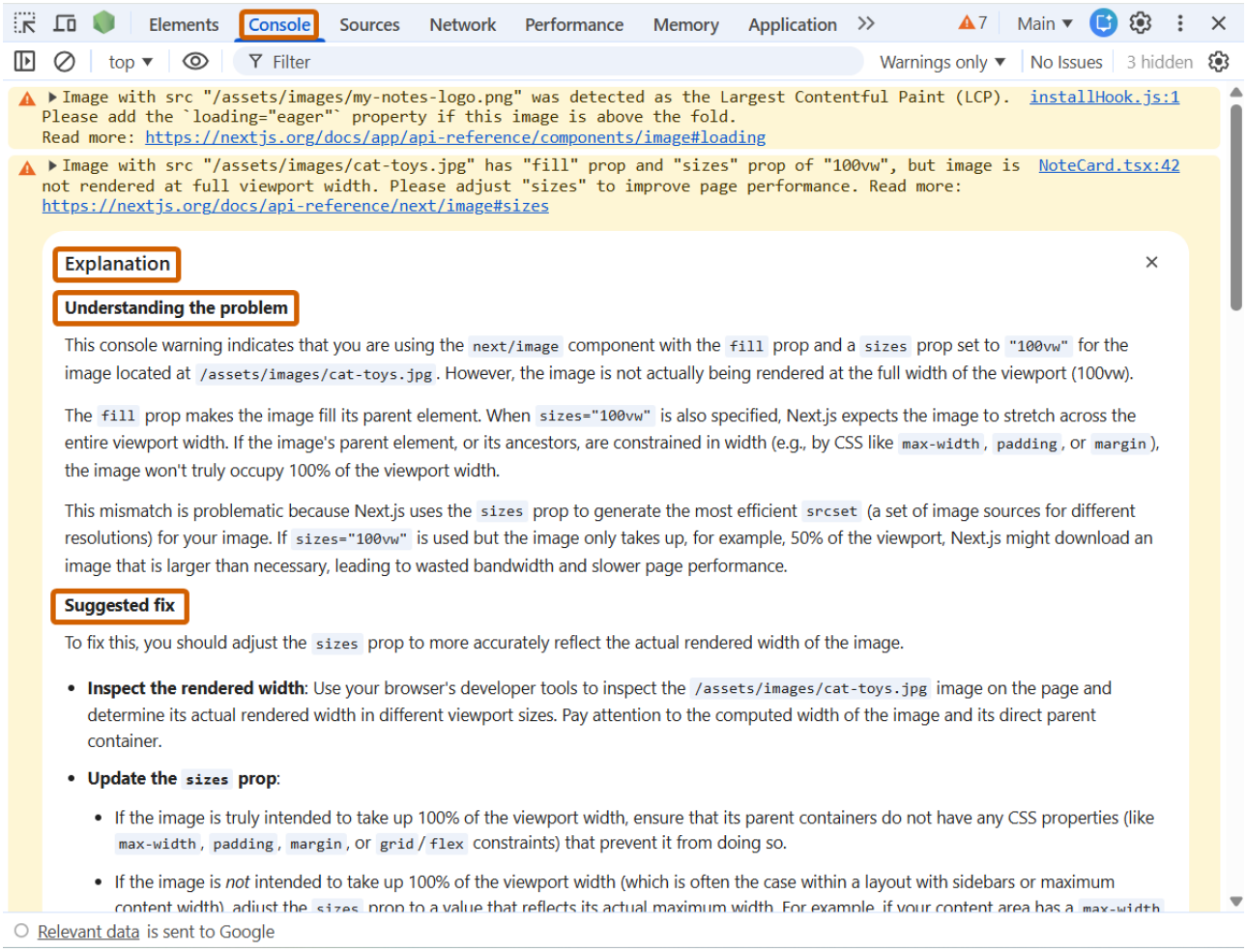
- Choose a specific message in the **Console**, hover over it, and then move your cursor over the **AI Console Insights** icon.



- It will show a label such as **Understand this warning**.



- Click it to generate an AI explanation:

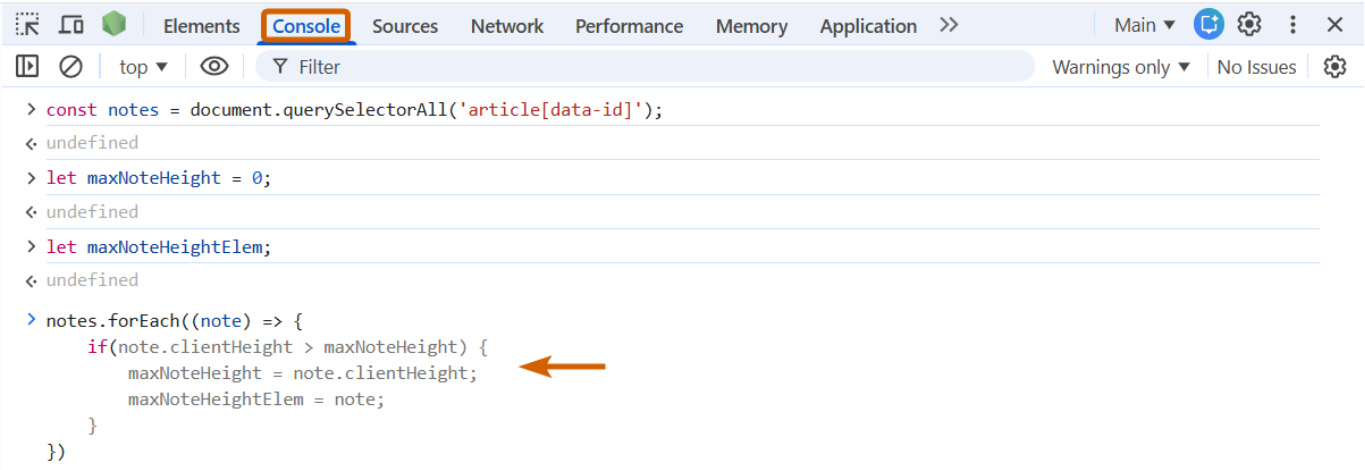


AI Innovations: Auto Annotations

The **Auto Annotations** feature allows you to automatically generate labels (titles) for performance trace annotations.
For more information, check the Performance Panel: Key Features and Workflow section.

AI Innovations: Code Suggestions

The **Code Suggestions** feature allows you to get auto-generated code as you type in the **Console** or **Sources** panels. You can press the `Tab` key to accept suggestions:



AI Assistance: Data and Privacy

Google's AI assistance tools in Chrome DevTools **send your data to Google's servers**. This includes:

- **Your prompts** and conversation history.
- **Web page context**, such as console logs, network requests and DOM content.

Because of this, you should **be extremely careful** when working on private or sensitive projects. **Never enter confidential or personal information** into the prompts.
The data is used by Google to improve its products and may be reviewed by humans. It is a best practice to avoid using this tool with any sensitive information.

> **Important Notes:**
>
> - **AI Innovations is an experimental feature:** It is evolving with each Google Chrome release. The functionality and UI may change as the Chrome team refines the feature. For the most up-to-date information, check the official documentation here: goo.gle/devtools-ai-reqs.
> - **AI can make mistakes:** As with any AI tool, it is not advisable to rely entirely on AI responses. Make sure to analyze and review AI-generated responses to benefit from them.

AI Assistance: Common Use Cases

- **Explain CSS and Styling**: Ask why an element is styled a certain way, how it interacts with other elements or get suggestions for CSS fixes.
- **Analyze Network Requests**: Understand why a network request failed, where it originated or why it's taking a long time.
- **Create a quick and detailed bug report**: If an error happens during the debugging process, like an API failure in the Network panel or console warnings, you can ask **AI Assistance** to create a quick and detailed bug report for that case.
- **Understand Source Code**: Ask questions about a specific file or code block to learn more about its purpose and functionality.
- **Optimize Performance**: Get suggestions for improving performance based on a recorded profile in the **Performance** panel.
- **Auto-generate Labels for Performance Annotations**: Quickly auto-generate labels for performance trace annotations based on a recorded profile in the **Performance** panel, which will help you gain an essential initial understanding of the performance trace entries.

2. Browser Debugger

Modern browsers include built-in developer tools with a JavaScript debugger (Chrome, Edge, Firefox, Safari). The core debugging concepts are the same across browsers: setting breakpoints, stepping through code, inspecting variables/scopes and pausing execution.

Browser Debugger: The `debugger;` Keyword

The `debugger;` keyword is a built-in JavaScript feature that acts as a programmatic breakpoint. By placing it in your code, you can intentionally pause code execution at a specific point and launch the browser's debugger, allowing you to inspect the state of your application in its live environment.

Simple Workflow

1. **Add the Breakpoint**: Place the `debugger;` statement in your JavaScript code where you want execution to pause.

```
editNote: async (note: Note): Promise<ActionStatus> => {
    debugger;

    set({ isNoteUpdateLoading: true });

    try {
      // Your main logic here
    } catch(error: unknown) {
      // Your error logic here
    }
}
```

2. **Open DevTools**: Open your browser's developer tools. **The debugger will only pause if DevTools is open**.
3. **Trigger the Code**: Perform the action in your application that runs the code containing the `debugger;` statement (e.g., click a button, edit a note).
4. **Execution Pauses**: The browser will automatically switch to the **Sources** panel, highlight the `debugger;` line and pause the script. You can now inspect variables, the call stack and more.

Conditional Debugging

A major advantage of using the `debugger;` keyword is the ability to easily create conditional breakpoints directly in your code. This is extremely efficient for finding bugs that only appear under specific circumstances.

Instead of pausing every time, you can wrap it in an `if` statement.

Example: Imagine you have a bug that only occurs when processing a note with a specific id.

```
editNote: async (note: Note): Promise<ActionStatus> => {
  if (note.id === '54') {
    debugger;
  }

  set({ isNoteUpdateLoading: true });

  try {
    // Your main logic here
  } catch(error: unknown) {
    // Your error logic here
  }
}
```

This saves you from manually stepping through hundreds of loop iterations to find the one problematic case.

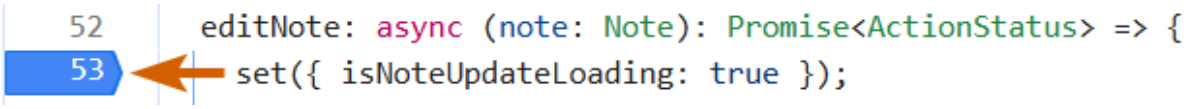
Browser Debugger: Adding Breakpoints: Chrome Example

In addition to the `debugger;` keyword, you can add breakpoints directly in the browser DevTools **Sources** panel. Each browser maintains this functionality in different ways. In this section, we will explore an example of adding breakpoints directly in the browser using Google Chrome.

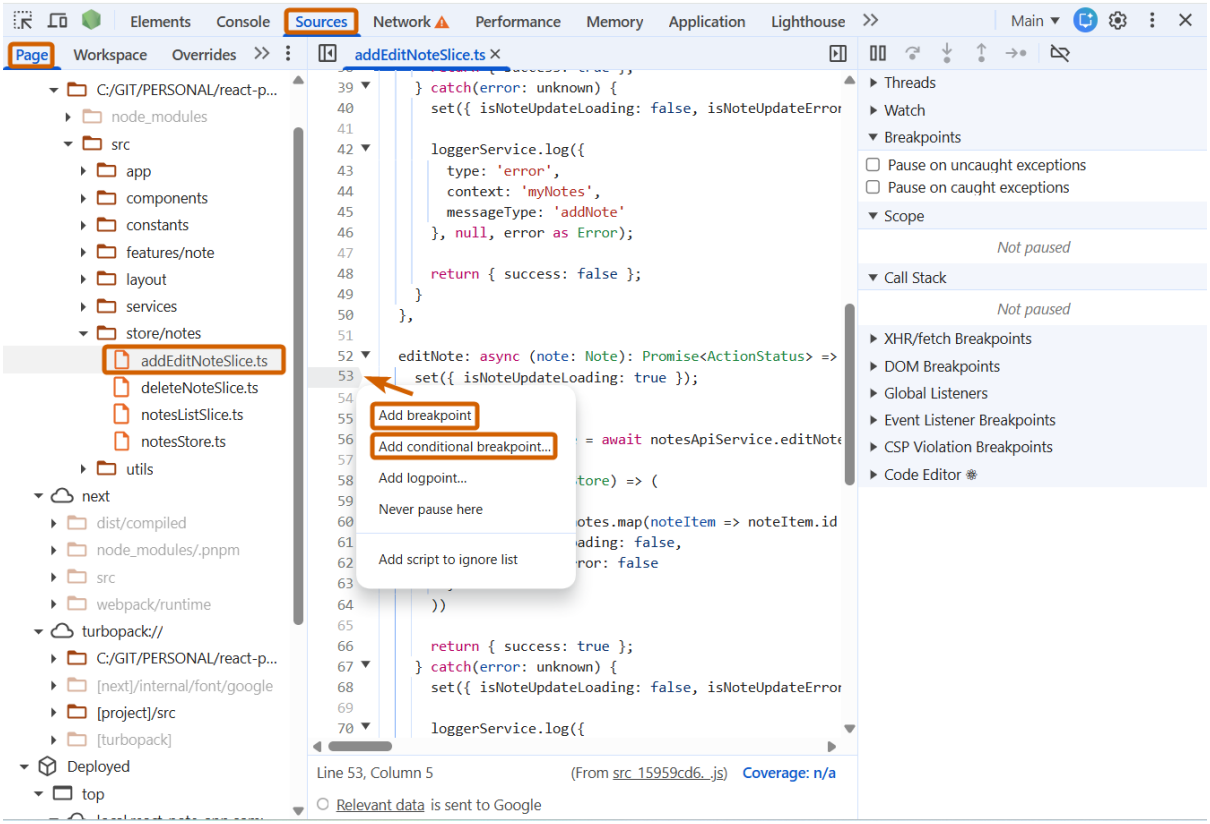
Adding Breakpoints in the Chrome Browser

Below are steps to follow to add a breakpoint in the Chrome browser:

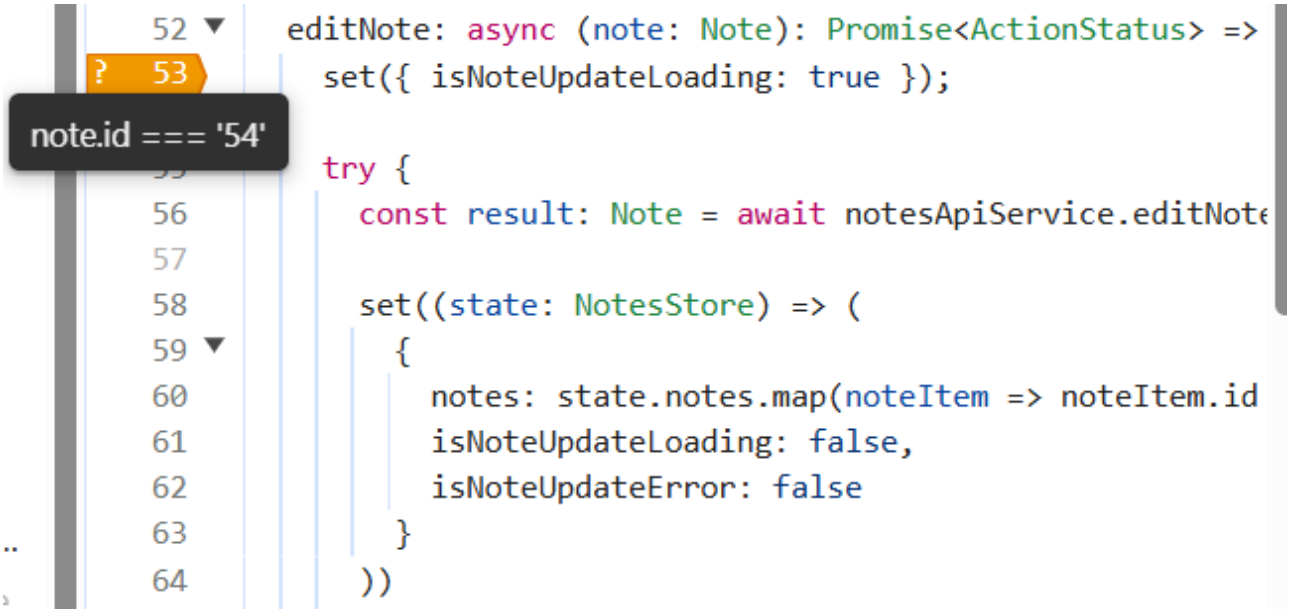
1. Navigate to your file in the **Sources** panel.
    - The simplest way to add a breakpoint is to **left-click** on a line number. A blue marker will appear.



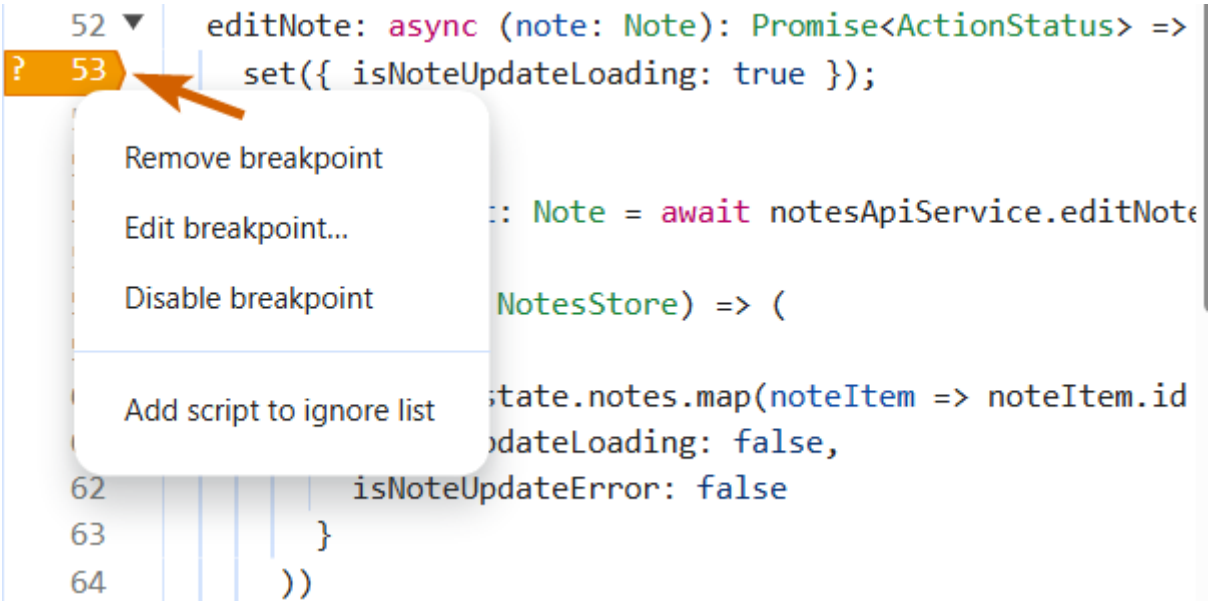
    - Alternatively, you can **right-click** the line and choose **Add breakpoint**.



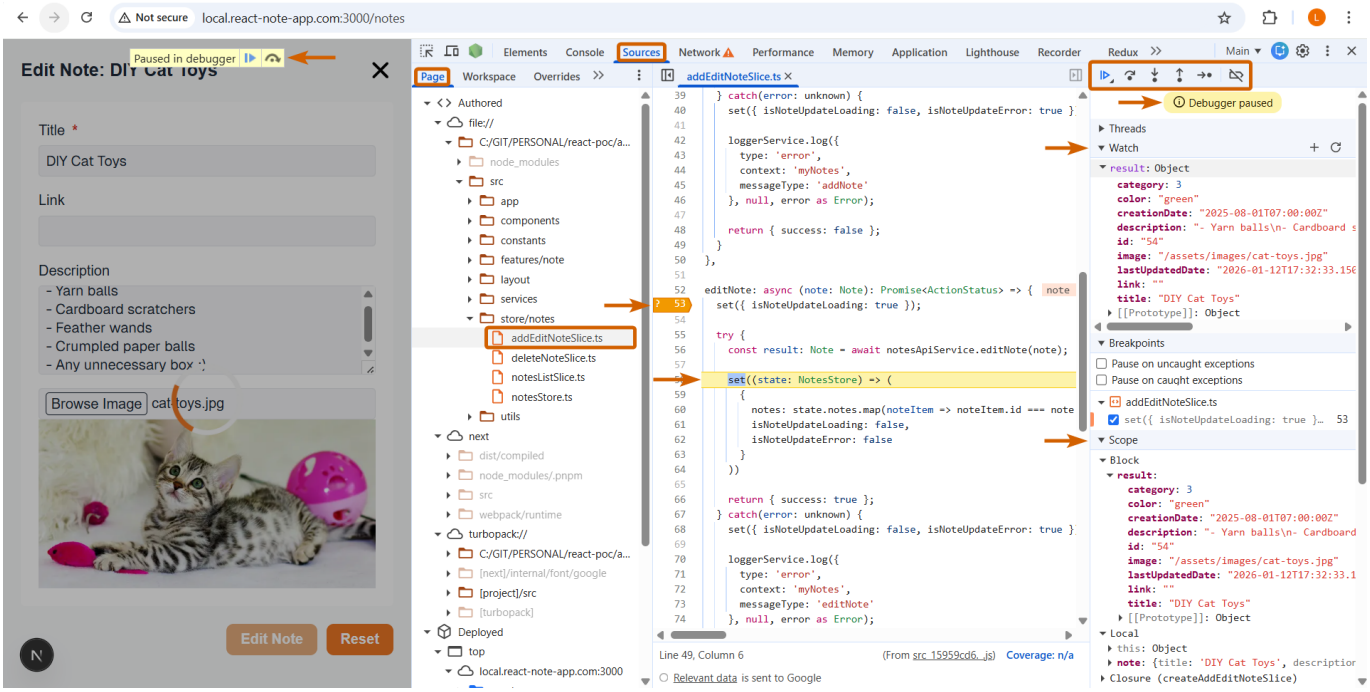
2. You can also create a **conditional breakpoint** that only pauses when a specific condition is true. Right-click the line number, select **Add conditional breakpoint...** and enter an expression (e.g., `note.id === 'specific-id'`).
   Conditional breakpoint markers are typically **orange**, while normal breakpoints are **blue**.



3. Manage existing breakpoints by **right-clicking** the breakpoint marker to **edit**, **disable** or **remove** them from the context menu.



4. Once your breakpoints are set, refresh the page or perform the action that triggers the code. Execution will pause at your breakpoint, allowing you to **inspect variables and continue debugging using the navigation controls**.



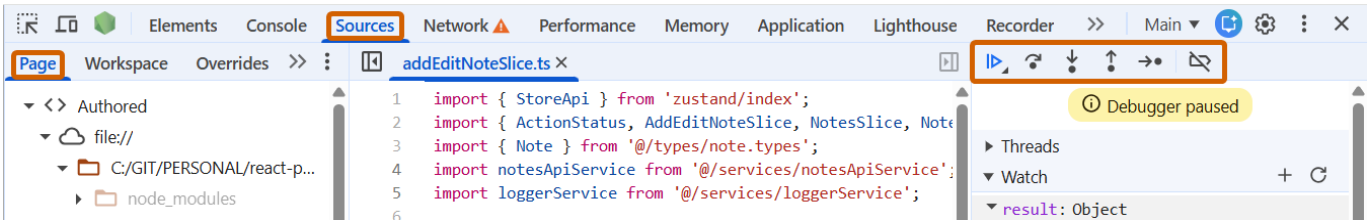
Navigating with Debugger Controls

Once paused, you can use the **Debugger Controls** toolbar (or keyboard shortcuts) in the **Sources** panel to navigate through your code:

- ▶ **Resume script execution (Windows/Linux/macOS: `F8`)**: Continues running the code until the next breakpoint is hit or the script finishes.
- ↷ **Step over next function call (Windows/Linux/macOS: `F10`)**: Executes the currently highlighted line and moves to the next one. If the line contains a function call, it will execute the entire function without stepping inside it.

- ↓ **Step into next function call (Windows/Linux/macOS: `F11`)**: If the highlighted line is a function call, this will move the debugger to the first line inside that function, allowing you to debug it line-by-line.
- ↑ **Step out of current function (Windows/Linux/macOS: `Shift+F11`)**: Steps out of the current function and returns to the calling function, to the line right after where it was called.
- → **Step (Windows/Linux/macOS: `F9`)**: The most granular step command. Simply points to the next line of code without any skips. It enters all function calls, including the browser's internal asynchronous operations (like the internals of `await`). It's rarely used for application debugging.

> **Note for macOS users:** Based on your settings, to trigger the shortcuts involving F-keys, you may need to hold the `Fn` key (e.g., `Fn+F8`).



Browser Debugger: Common Use Cases and Advantages

The primary use case for the Browser Debugger feature is the **quick and direct inspection of client-side code**. It is a powerful feature which can help you to save countless hours while debugging your client-side code compared to inserting numerous `console.log` commands into your code. It's the best approach for **debugging UI rendering, DOM events, client-side state and hydration issues**.

Below are advantages of using the **Browser Debugger** feature:

- **Real Execution Environment**: Runs in the real execution environment (the browser's JS engine).
- **Full Environment Access**: Allows full access to the live DOM, `window` object and all browser APIs.
- **DevTools Integration**: Provides direct integration with other panels in the rich DevTools ecosystem (Network, Performance, Application, etc.).
- **Extension Support**: Provides access to your installed browser extensions (e.g., React DevTools and Redux DevTools).
- **Advanced Editing**: Supports conditional breakpoints and live code editing directly in the **Sources** panel.
- **Speed and Low Overhead**: Often faster for purely client-side tasks than an IDE debugger. This is especially true when the IDE is managing a complex, multi-layer session (like a Next.js app with client, server and API debuggers attached at once), which adds significant overhead.
- **Zero Setup and Quick Learning Curve**: Works instantly without any configuration files (e.g., `launch.json` in Cursor IDE). You can open DevTools and set a breakpoint in seconds, making it very approachable and fast to start with.

3. IDE Debugging

Setting up a debugger in your favorite IDE (Integrated Development Environment) provides a powerful and flexible way to inspect your front-end applications. It is especially useful for modern full-stack frameworks like **Next.js** or SSR-based front-end setups like an **Angular SSR** application, where you often need to debug the client-side UI, the server-side rendering (SSR) layer and a separate API layer all at once.

An IDE debugger gives you a unified place to trace issues as they move across different layers of your application, including the server-side, client-side and API.

3.1 Debugging in WebStorm

WebStorm is a dedicated IDE for JavaScript, TypeScript and web frameworks (e.g., React, Angular and Vue) powered by **JetBrains**. It accelerates development with intelligent autocomplete, real-time error detection, Git integration and built-in AI assistance.

WebStorm provides these features out of the box. Unlike editors like Cursor (VS Code-based), where you often need to manually install extensions for advanced capabilities, WebStorm comes pre-configured, allowing you to focus immediately on development and debugging rather than setup.

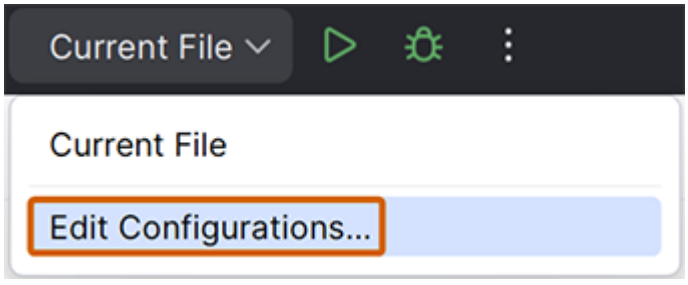
Debugging in WebStorm: Full-stack Next.js Setup

Let's walk through setting up WebStorm debugging for a complex **Next.js** project that includes a separate **Express.js mock API layer**.

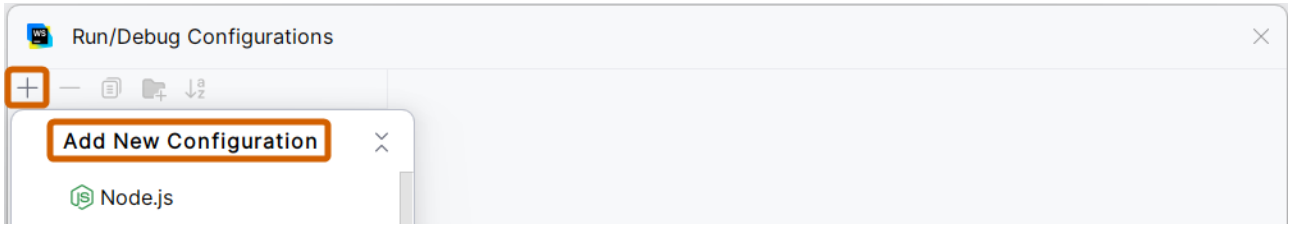
1. Full-stack Next.js Setup: Back-end (Server-side Code) Debugger Setup

This configuration runs the Next.js server itself in debug mode.

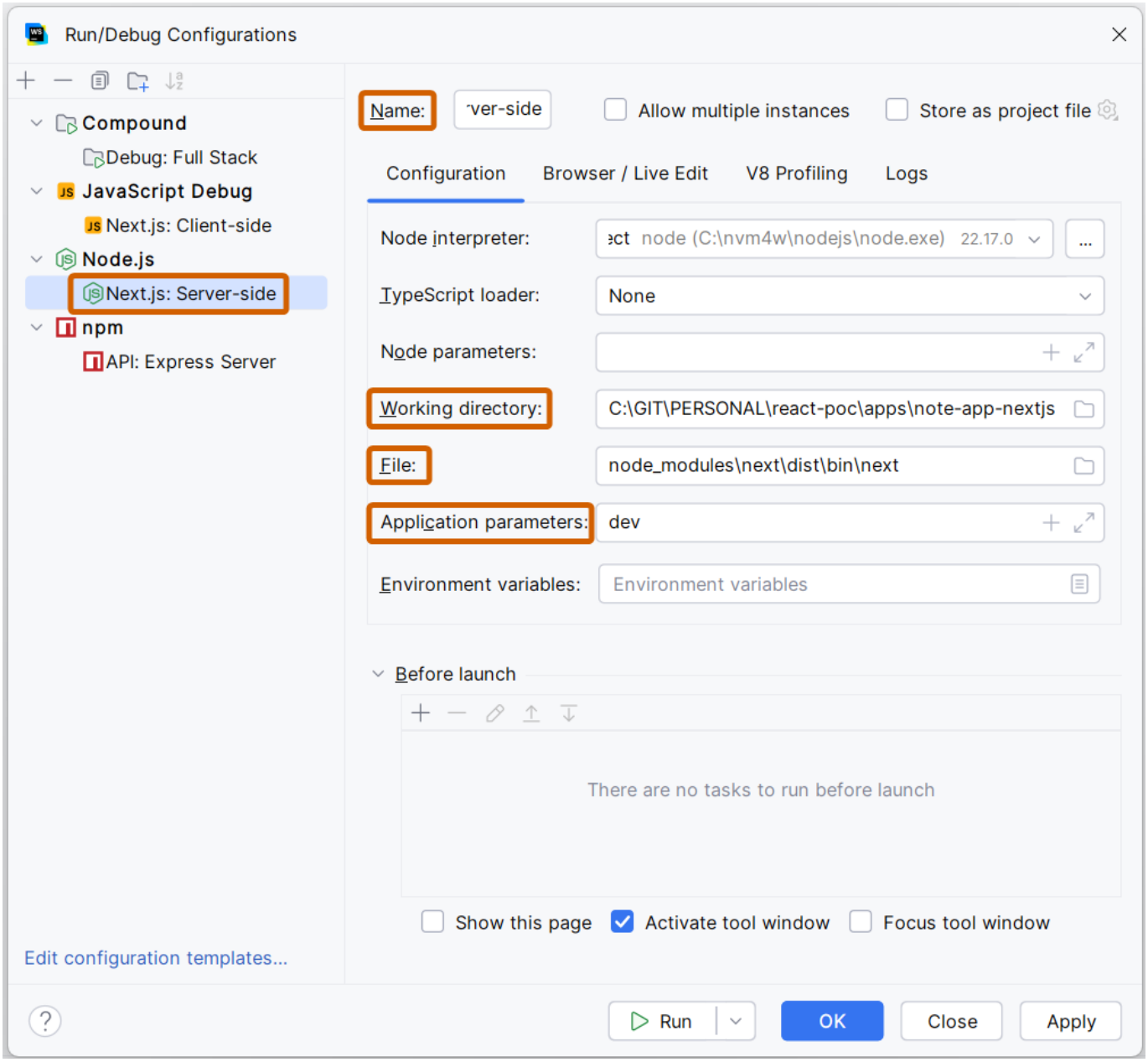
- Click the **Edit Configurations...** option from the **Run/Debug** dropdown.



- Click the **+** icon in the opened **Run/Debug Configuration** dialog.



- Choose **Node.js** in the opened **Add New Configuration** dropdown.
- Specify configuration details:
  - **Name**: `Next.js: Server-side` (this is just a label for your configuration, choose any name you prefer)
  - **Working directory**: The root folder path of your Next.js project.
  - **File**: The path to Next.js's executable: `node_modules\next\dist\bin\next`
  - **Application parameters**: The script command, typically `dev`.
- Click **OK** or **Apply** to save the changes.

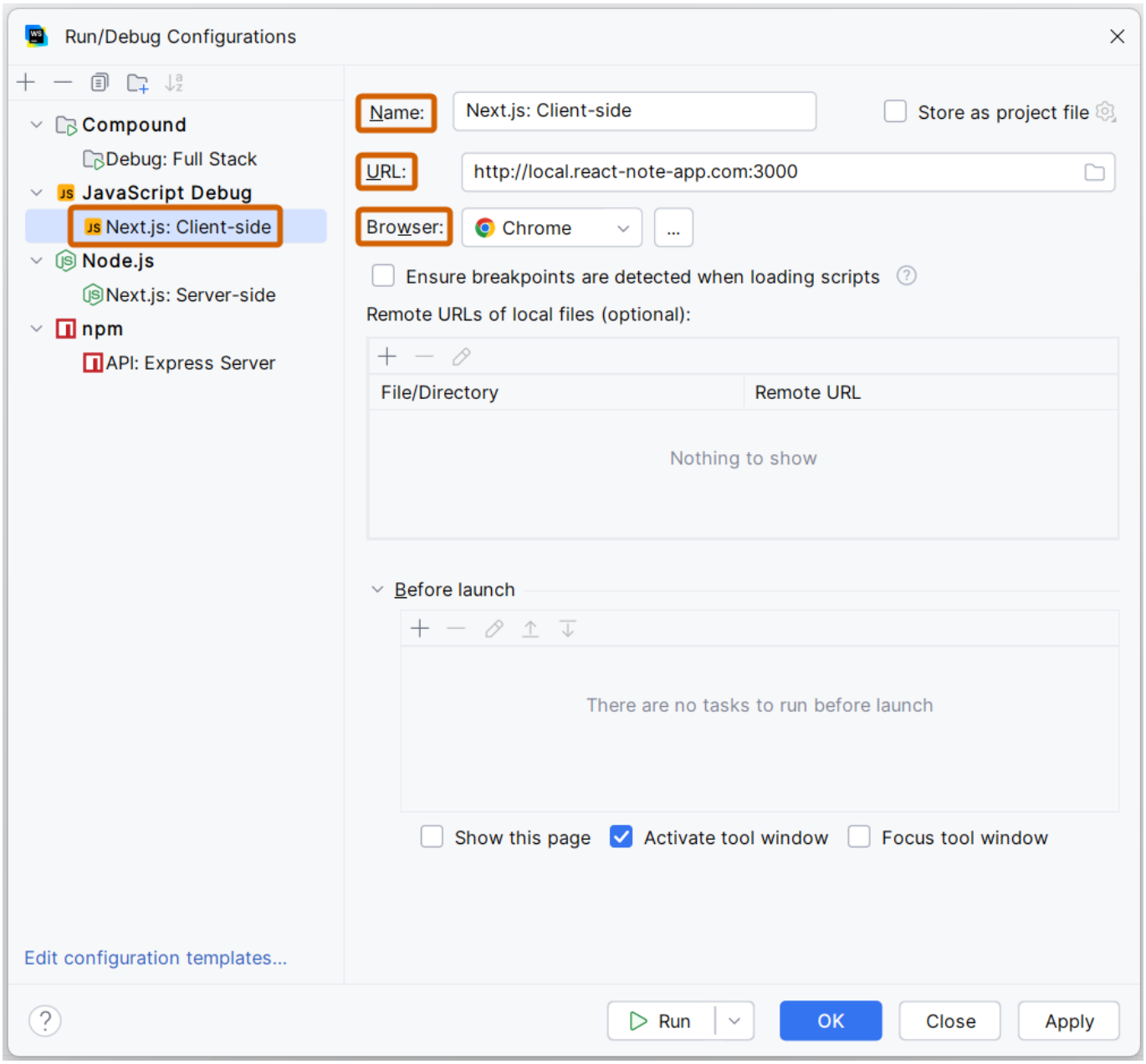


2. Full-stack Next.js Setup: Client-side (Browser) Debugger Setup

This configuration attaches the IDE's debugger to the browser.

- Click the **+** icon and choose **JavaScript Debug** in the opened **Add New Configuration** dropdown.
- Create a configuration with the following details:
  - **Name**: `Next.js: Client-side`.

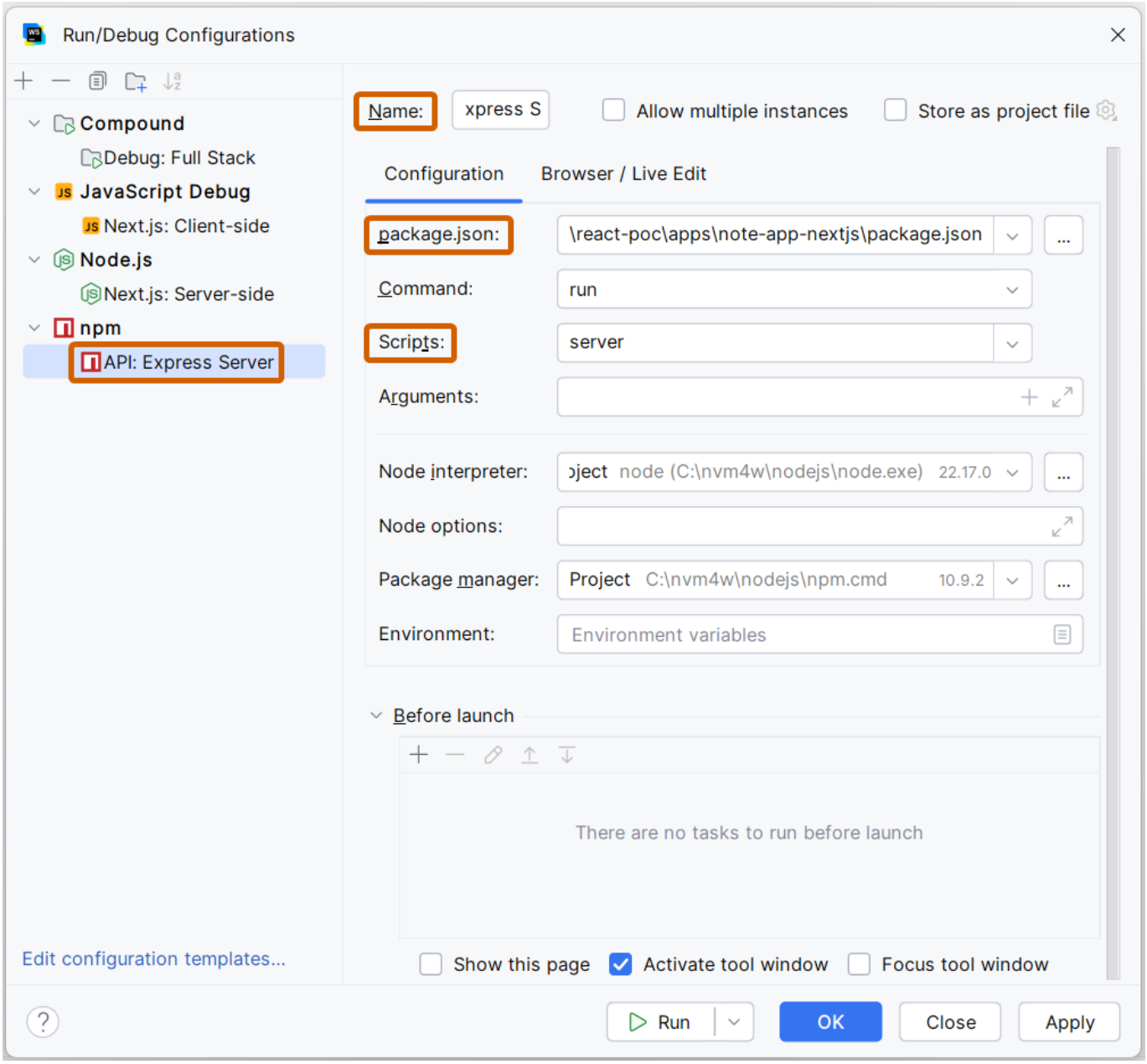
- **URL**: The URL of your running local application (e.g., `http://local.react-note-app.com:3000`).
- **Browser**: Choose your preferred browser, like Chrome or Edge.



3. Full-stack Next.js Setup: Express.js (Mock API Server) Debugger Setup

This configuration runs your separate API layer.

- Click the **+** icon and choose **npm** in the opened **Add New Configuration** dropdown.
- Create a configuration with the following details:
  - **Name**: `API: Express Server`.
  - **package.json**: The path to your project's `package.json`.
  - **Scripts**: The script name from `package.json` that runs the server (e.g., `server`).

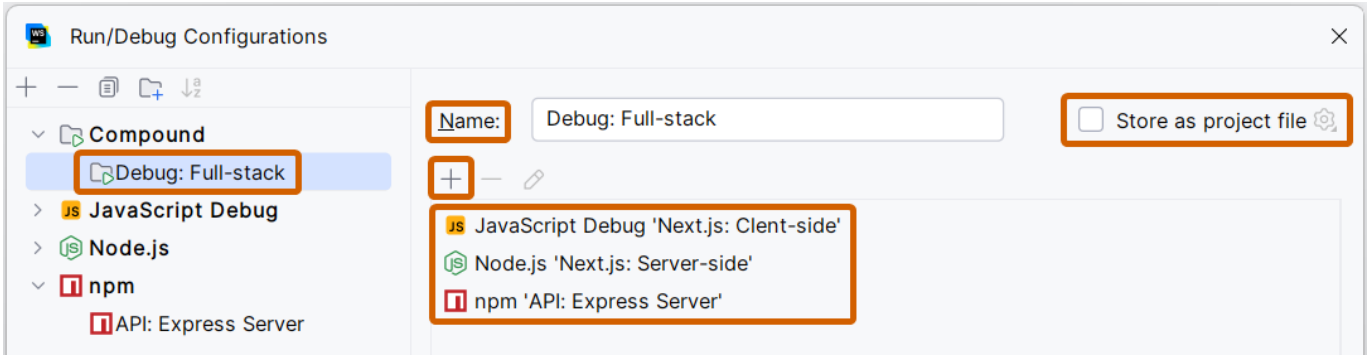


4. Full-stack Next.js Setup: Compound Debugger Setup (Debugging All Layers at Once)

This compound configuration launches all your debuggers together and allows you to debug all layers (client, server, API) at the same time with a single command.

- Click the **+** icon and choose **Compound** in the opened **Add New Configuration** dropdown.
- Specify **Name** as `Debug: Full Stack`.
- Click the **+** icon within the dialog content to add the three configurations you just created: `Next.js: Server-side`, `Next.js: Client-side` and `API: Express Server`.

> **Note:** You can use the **Store as project file** checkbox for each configuration to save it as a file, allowing you to commit it to the repository or reuse it locally.



After adding this configuration, you can add breakpoints in all three layers and start the debugger using a single command (`Debug: Full Stack`). The debugger will stop in all layers, giving you a powerful, holistic debugging workflow for your Next.js full-stack application.

Debugging in WebStorm: Angular SSR Setup

When creating an Angular project using the `Angular CLI`, it already generates default debugging commands for WebStorm (`Angular Application` and `Angular CLI Server`). While you can use these to debug your client-side and server-side code independently, you need additional setup to achieve a unified debugging flow for both client and server sides in an SSR-based Angular project.

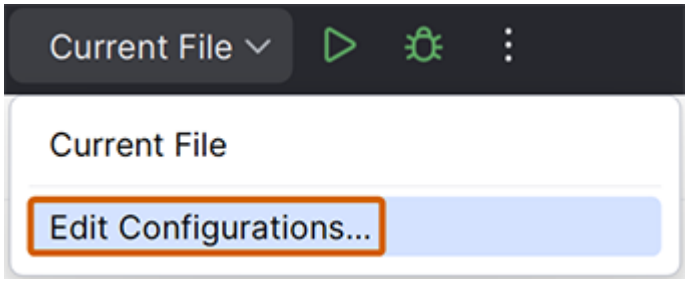
Since `ng serve` also handles the SSR (Server-Side Rendering) layer, we can build the debugging configuration logic on top of running `ng serve` and attaching a browser debugging session to it. WebStorm handles this workflow seamlessly.

Let's walk through setting up WebStorm debugging for an SSR-based Angular project.

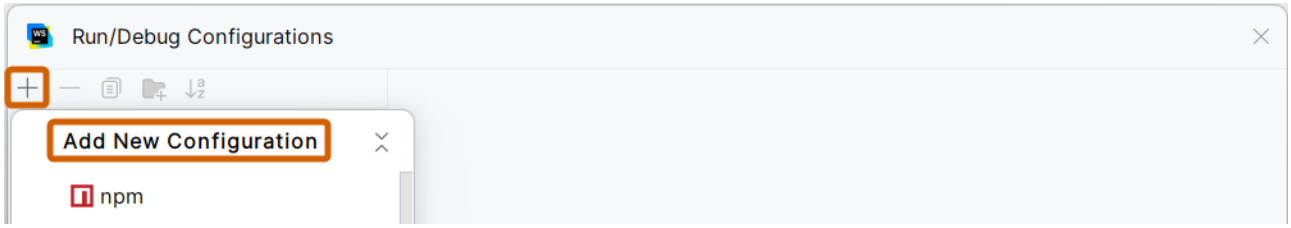
1. Angular SSR Setup: Angular CLI Server Debugger Setup

This configuration runs `ng serve` in debug mode. There is no need to change anything in the default configuration created by the Angular CLI. However, if needed, below are instructions on how to manually configure it:

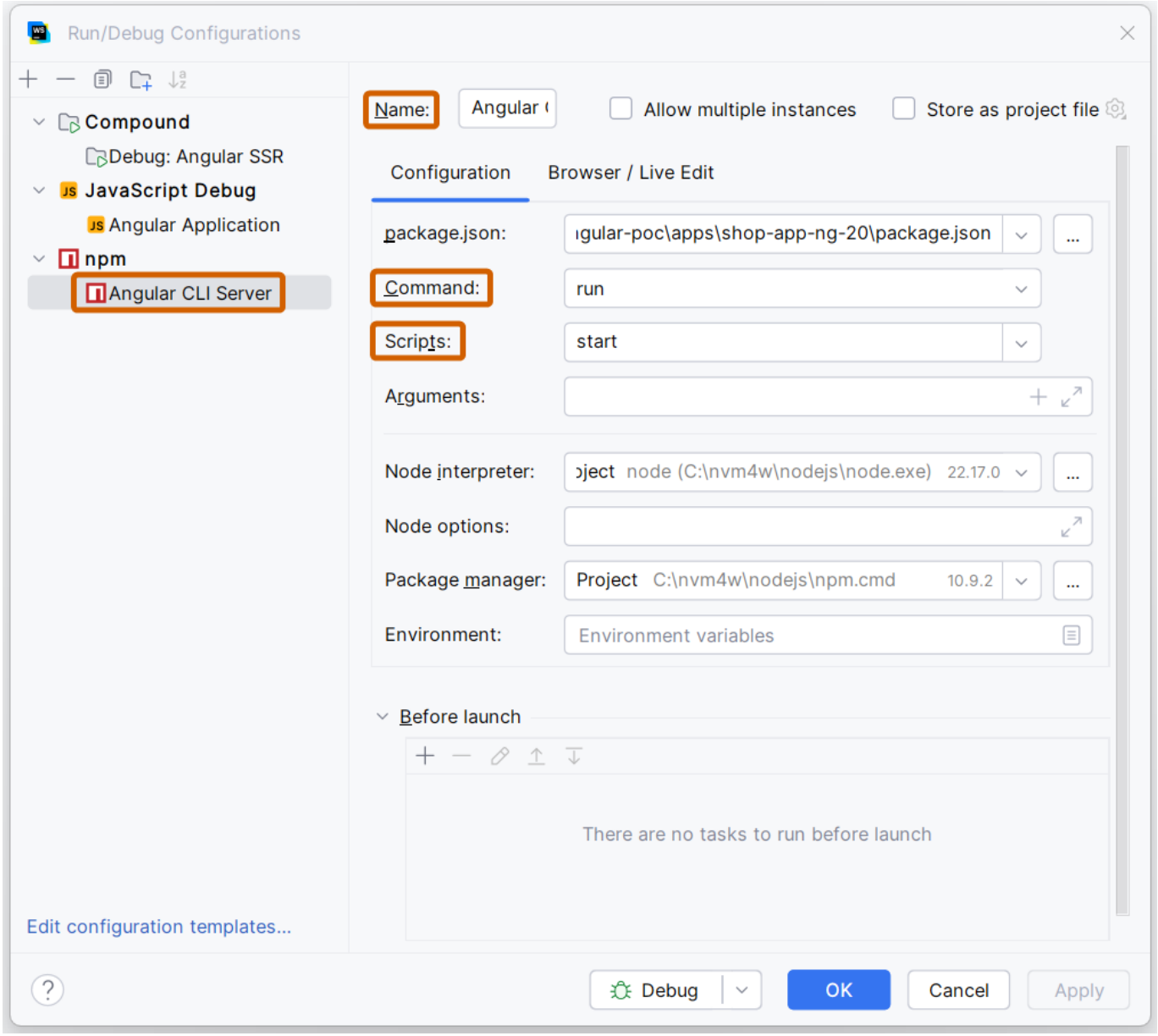
- Click the **Edit Configurations...** option from the **Run/Debug** dropdown.



- Click the **+** icon in the opened **Run/Debug Configuration** dialog.



- Choose **npm** in the opened **Add New Configuration** dropdown.
- Specify configuration details:
  - **Name**: `Angular CLI Server`
  - **Command**: `run`
  - **Scripts**: `start` (This is the script in your `package.json` which initiates `ng serve`)
- Click **OK** or **Apply** to save the changes.



2. Angular SSR Setup: Angular Application (Client-side) Debugger Setup

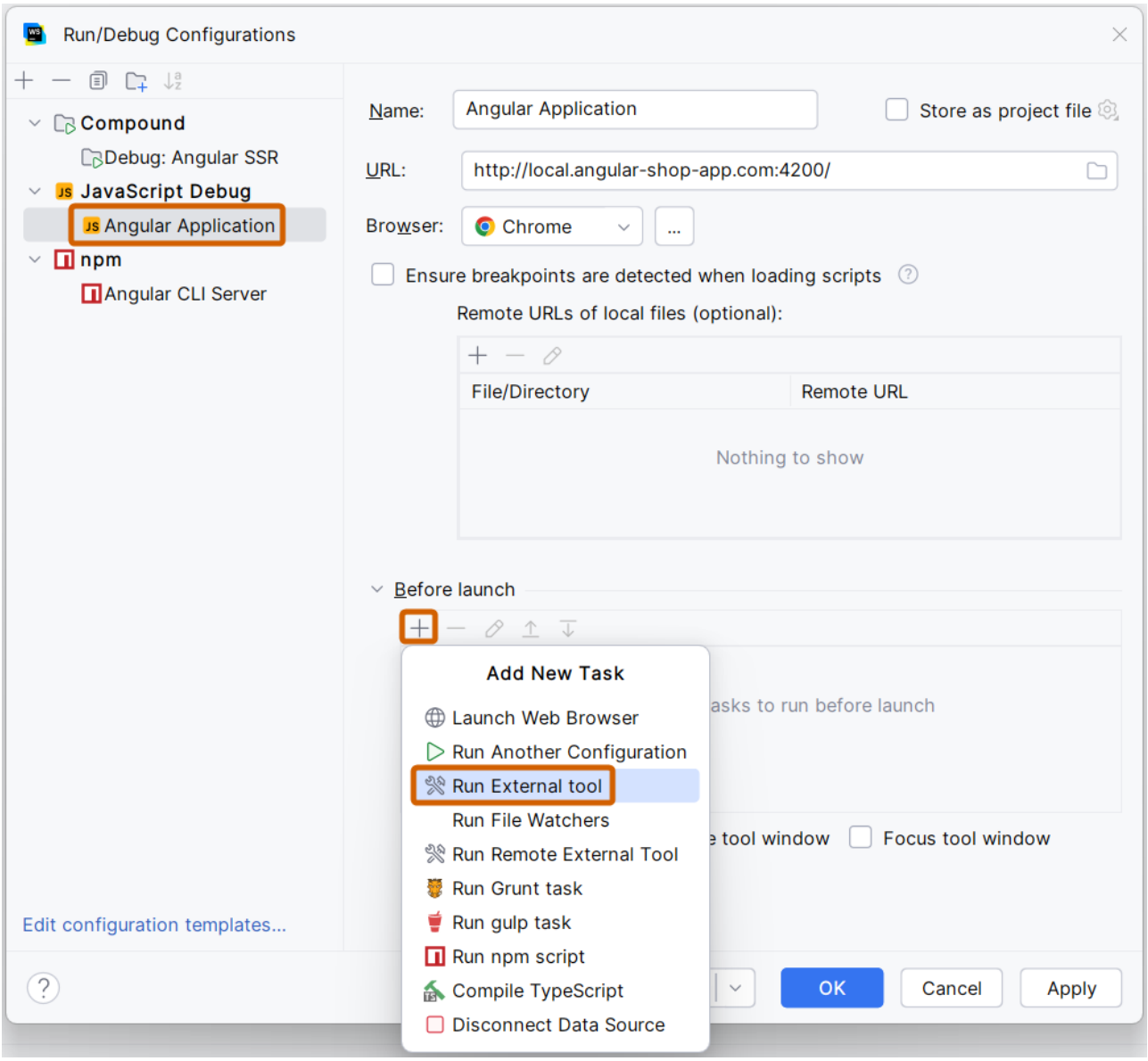
This configuration attaches the IDE's debugger to the browser. It is also created by the Angular CLI by default. However, you need to make slight modifications to make it work as part of a full-stack debugging workflow. In particular, you need to add an external tool that waits for the necessary port to become available, meaning `ng serve` has completed its startup process, and only then open the debugging window.

Step A: Below are the steps to attach the IDE's debugger to the browser (this configuration is created by the Angular CLI by default, but here is how to configure it manually):

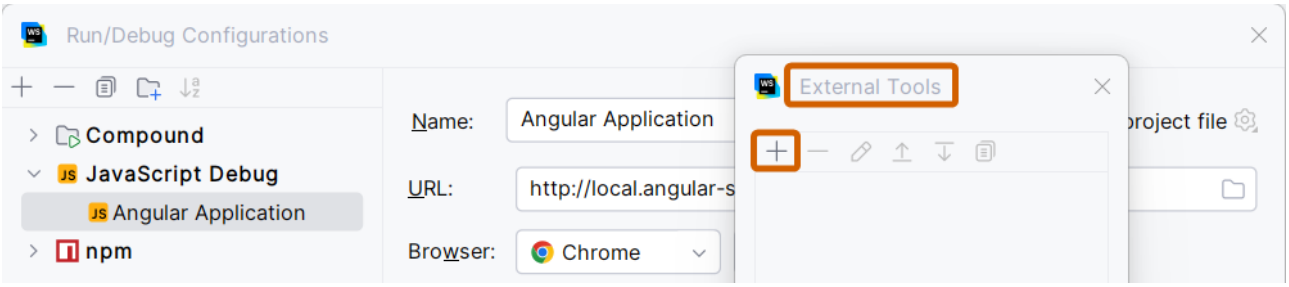
- Click the **+** icon and choose **JavaScript Debug** in the opened **Add New Configuration** dropdown.
- Create a configuration with the following details:
  - **Name**: `Angular Application` (this is just a label for your configuration, choose any name you prefer)
  - **URL**: The URL of your running local application (e.g., `http://local.angular-shop-app.com:4200`).
  - **Browser**: Choose your preferred browser, like Chrome or Edge.

Step B: This is the step to make the command wait until port `4200` is available before opening the debugging browser window.

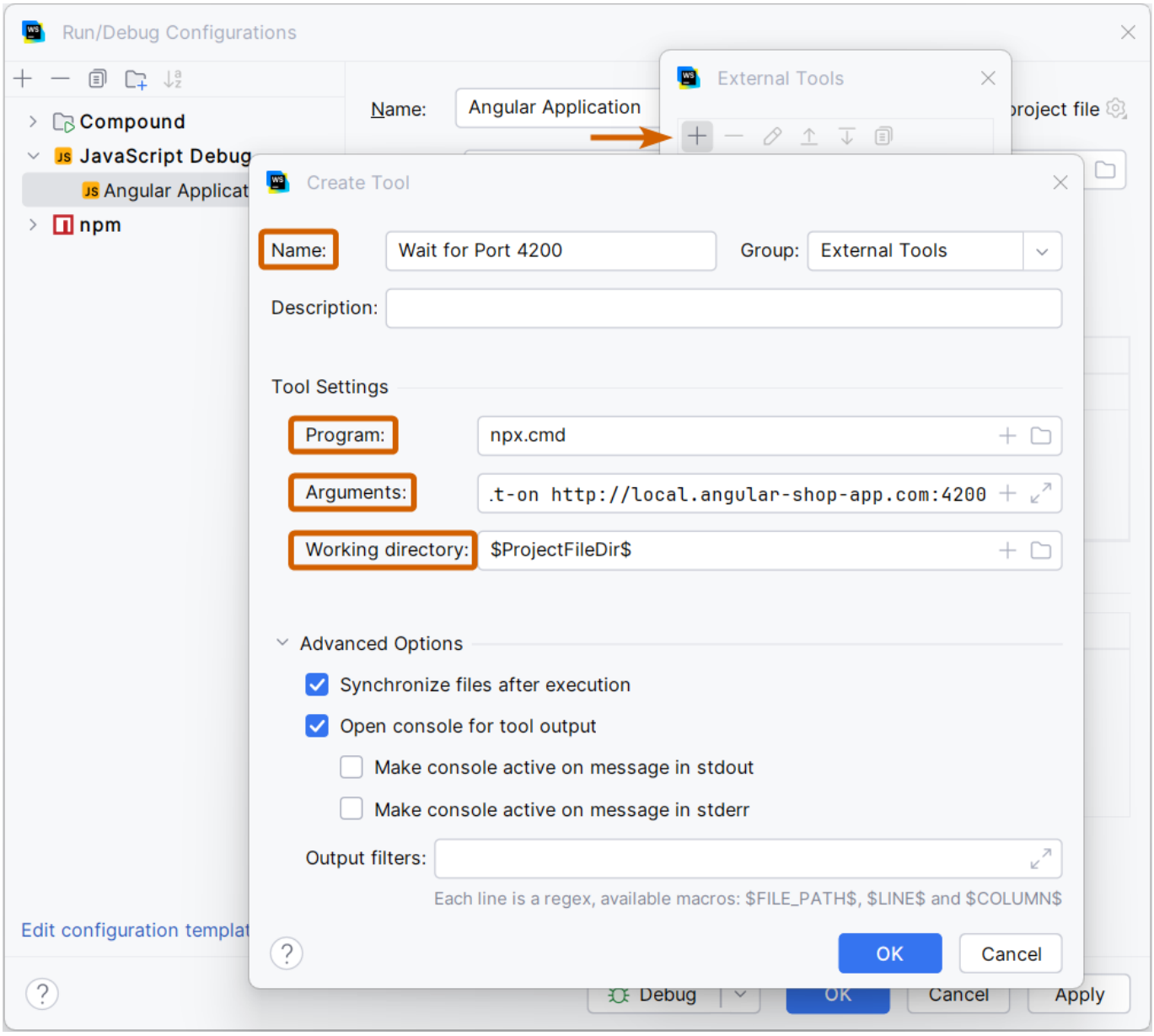
- Click the **+** icon in the **Before Launch** section in the dialog content and choose **Run external tool**.



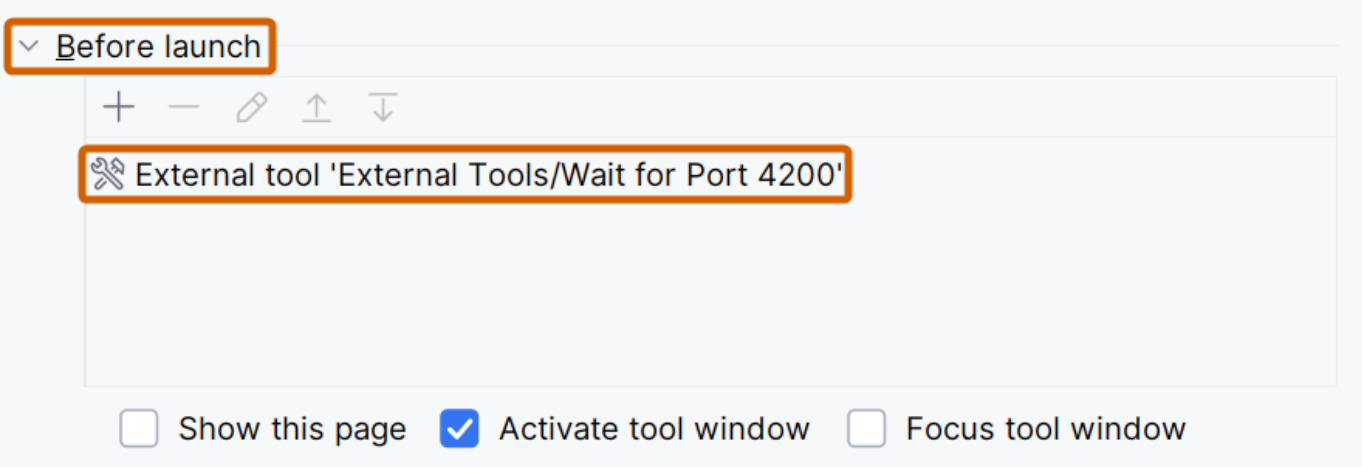
- In the opened **External Tools** dialog, click the **+** icon to add a new tool.



- In the opened **Create Tool** dialog, fill in the required details and click **OK** to save the result:
  - **Name**: `Wait for Port 4200`
  - **Program**: `npx.cmd` (Use `npx` for macOS and Linux)
  - **Arguments**: `wait-on http://local.angular-shop-app.com:4200` (Replace this URL with the URL of your running local application)
  - **Working Directory**: `$ProjectFileDir$`



The external tool will now appear in the **Before launch** section of the configuration.

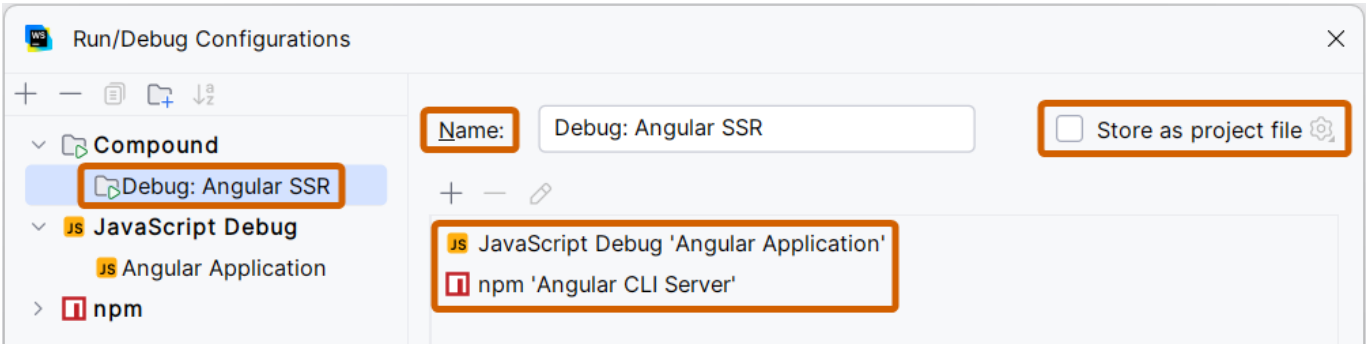


3. Angular SSR Setup: Compound Debugger Setup (Debugging All Layers at Once)

This is the compound configuration that launches all your debuggers together.

- Click the **+** icon and choose **Compound** in the opened **Add New Configuration** dropdown.
- Specify **Name** as `Debug: Angular SSR`.
- Click the **+** icon within the dialog to add the two configurations you just created: `Angular CLI Server` and `Angular Application`.

> **Note:** You can use the **Store as project file** checkbox for each configuration to save it as a file, allowing you to commit it to the repository or reuse it locally.



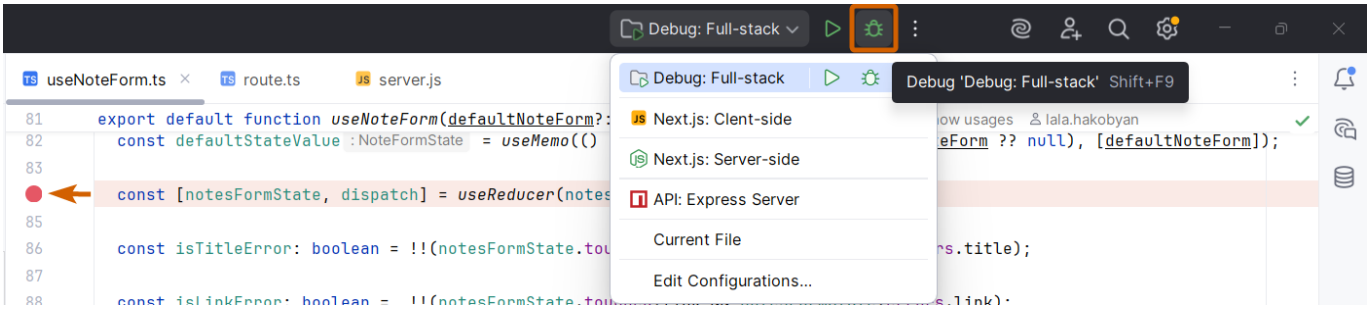
After adding this configuration, you can add breakpoints in both the server (e.g., inside `app.use` in `server.ts`) and the client-side code, and start the debugger using a single command (`Debug: Angular SSR`). The debugger will stop in both layers, allowing you to understand the SSR and hydration flow better and debug effectively.

Debugging in WebStorm: Starting Your Debug Session

1. **Set Breakpoints**: In your code, click the gutter (the area to the left of the line numbers). A **red dot icon** (🔴) will appear, indicating a breakpoint. You can set these in your client-side components, server-side code in Next.js, server files in your SSR-based Angular application and your Express.js API routes. You can also add or remove breakpoints while the debugging session is active.

2. **Launch the Debugger**: Select your compound configuration (e.g., **Debug: Full Stack** or **Debug: Angular SSR**) from the top dropdown and click the **debug icon ( )**.

3. **Debug**: WebStorm will launch all configurations included in your compound setup and open a new browser window. When your application's code hits one of your breakpoints, execution will pause and the **Debugger** window will appear, allowing you to inspect the state and navigate through the code using the debugger controls.



Debugging in WebStorm: Navigation Controls and State Inspection

Once the debugger is paused, you can use these tools to control the application execution flow and inspect its state:

Navigation Controls:

- ▶ **Resume Program (Windows/Linux: `F9` | macOS: `Cmd+Option+R`)**: Resumes execution until the next breakpoint is hit or the end of the program is reached.
- ↷ **Step Over (Windows/Linux/macOS: `F8`)**: Executes the currently highlighted line and moves to the next one. It does **not** enter any functions on the current line.
- ↓ **Step Into (Windows/Linux/macOS: `F7`)**: If the current line contains a function call, this will move the debugger to the first line *inside* that function.
- ↑ **Step Out (Windows/Linux/macOS: `Shift+F8`)**: Steps out of the current function and returns to the calling function, to the line right after where it was called.
- ⟳ **Rerun (Windows/Linux: `Ctrl+F5` | macOS: `Cmd+R`)**: Restarts the entire debugging session.
- ■ **Stop (Windows/Linux: `Ctrl+F2` | macOS: `Cmd+F2`)**: Ends the debugging session completely.

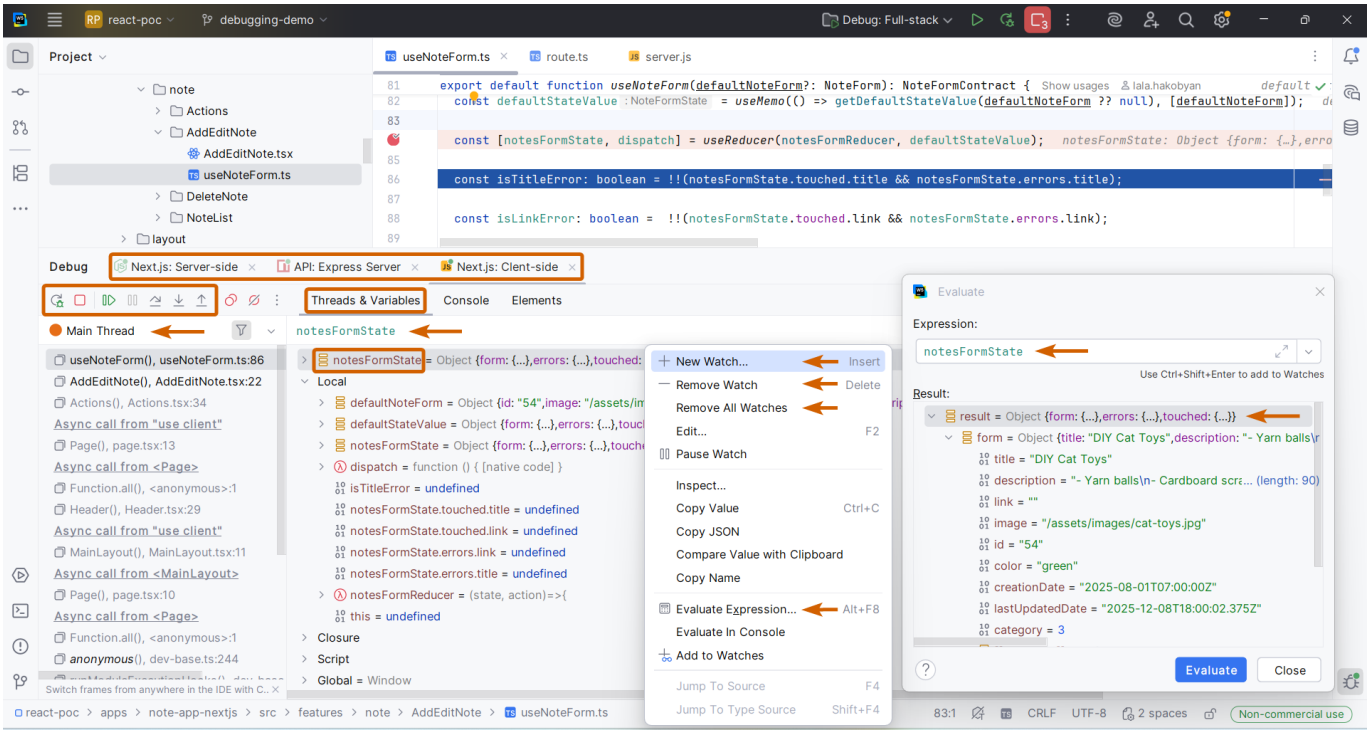
> **Note for macOS users:** Based on your settings, to trigger the shortcuts involving F-keys, you may need to hold the `Fn` key (e.g., `Fn+F8`).

Threads and Variables Inspection Panel: This panel allows you to inspect the current values of all variables in the scope.

- You can **watch** specific variables to track how they change. To add a new watch, type the desired property name in the input and select it.
  After adding a watch, you can right-click on it and remove it individually via the **– Remove Watch**

option. Alternatively, you can remove all watches via the **Remove All Watches** option and add a new watch via the + **New Watch...** option.

- You can right-click on a watched variable or any other variable and press **Evaluate Expression**, which will allow you to choose an expression on your desired property and evaluate it directly in the opened modal.
- On the left side, you can also select the desired **thread** (e.g., Main Thread) from the dropdown and review the current call stack for that thread.



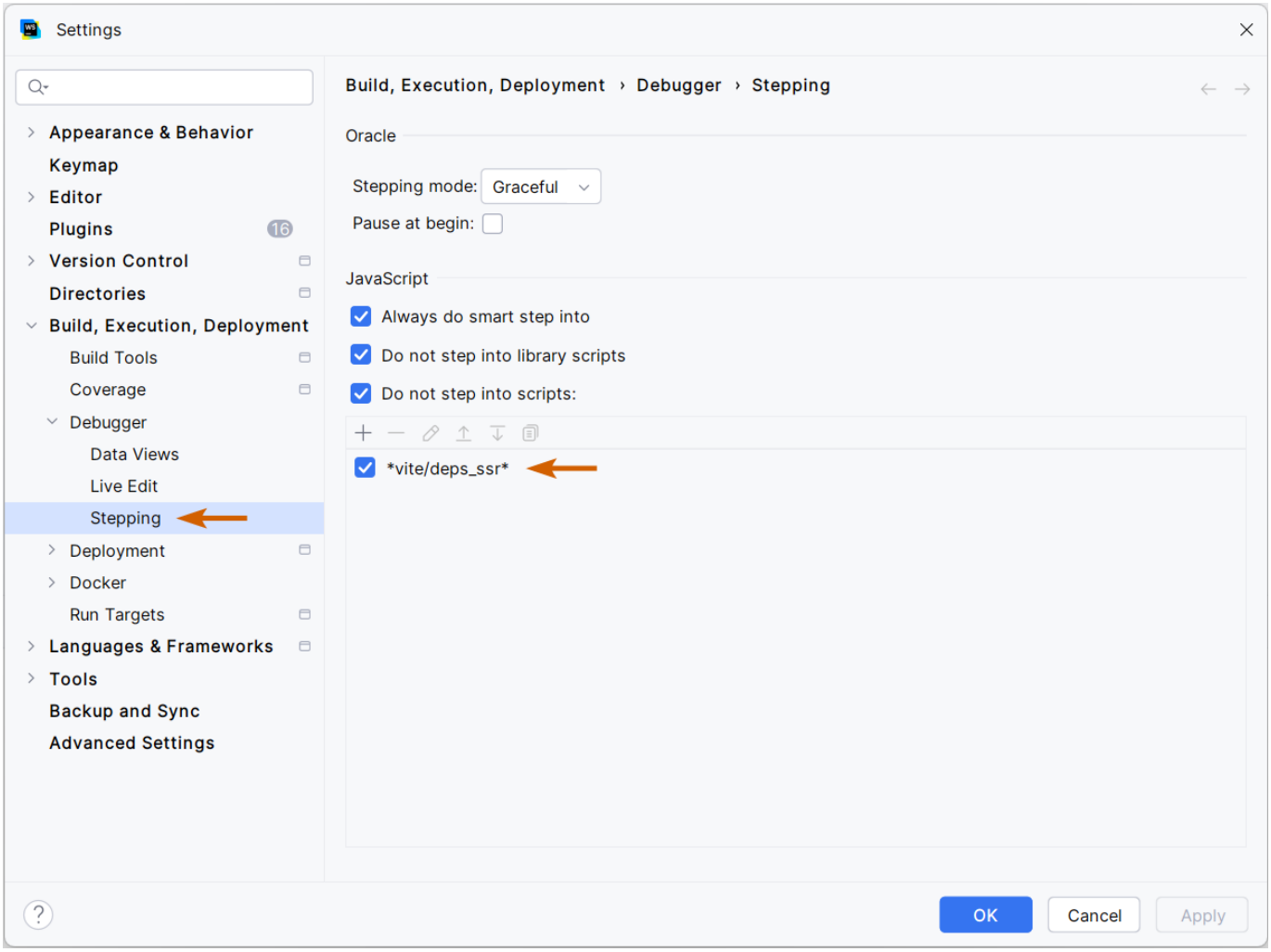
Debugging in WebStorm: Skip Files

To prevent the debugger from entering specific internal files, you can add them to the exclusion list in **WebStorm Settings**.

Below are the steps to follow:

1. Open **File > Settings**.
2. Navigate to **Build, Execution, Deployment > Debugger > Stepping**.
3. Check the box **Do not step into library scripts**.
4. Check the box **Do not step into scripts**, click the + and add the path pattern to skip.
   For example, to skip internal Vite/Angular files, use the pattern: `*vite/deps_ssr*`.
5. Click **Apply** or **OK** to save the changes.

Restart your IDE if needed for the changes to take effect.



3.2 Debugging in Cursor

Cursor is a powerful, AI-powered code editor forked from Visual Studio Code. As a result, it shares the same robust debugging architecture and uses a `.vscode/launch.json` file to configure debug sessions, just like in VS Code.

Similar to WebStorm, setting up Cursor for your complex application, whether it is a full-stack Next.js application or an Angular SSR-based application, allows you to debug client-side code, server-side rendering and any additional API layers within a single, unified workflow.
However, unlike WebStorm, Cursor doesn't come with built-in tools, so you need to configure them manually. While in WebStorm you could use the UI wizard to set up your debugging flow, in Cursor, to enable debugging, you need to create a `launch.json` file inside a `.vscode` folder at the root of your project.

Debugging in Cursor: Full-stack Next.js Setup

Let's walk through setting up Cursor debugging for a complex **Next.js** project that includes a separate **Express.js mock API layer**.

The server-side configuration below differs from what the official Next.js documentation often suggests. The official documentation may recommend a `"request": "launch"` configuration. However, with that

method, breakpoints in server-side code can sometimes become **unbound** (unverified) because the debugger may struggle to find the correct source maps for the Node.js process.
A more reliable method is to use a `"request": "attach"` configuration. This involves starting the Next.js development server first and then attaching the debugger to the already-running underlying Node.js process.

Below is a 4-step guide for a full-stack, three-layer debugging setup for a Next.js application:

1. Create your `launch.json` file

Set up the following configurations:

- **Next.js: Server-side: Attach to Process**: Attach debugger to underlying Node.js process. The port should match the inspected Node.js port which you can find in the terminal after running the development server command (`npm run dev`) with `--inspect`.
- **Next.js: Client-side**: Launch the browser (e.g., Chrome) with your local development server URL (e.g., `http://local.react-note-app.com:3000` or `http://localhost:3000`)
- **API: Express Server**: Launch your Express.js API server with `npm` (`npm run server` is the command that launches the server).
- **Debug: Full Stack:** Compound configuration to launch all three layers together (client-side, server-side and API)

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Next.js: Server-side: Attach to Process",
      "type": "node",
      "request": "attach",
      "port": 9230
    },
    {
      "name": "Next.js: Client-side",
      "type": "chrome",
      "request": "launch",
      "url": "http://local.react-note-app.com:3000",
      "webRoot": "${workspaceFolder}",
      "skipFiles": [
        "${workspaceFolder}/node_modules/**",
        "${workspaceFolder}/.next/**",
        "<node_internals>/**"
      ]
    },
    {
      "name": "API: Express Server",
      "type": "node-terminal",
      "request": "launch",
      "command": "npm run server",
```

```
      "cwd": "${workspaceFolder}",
      "skipFiles": [
        "${workspaceFolder}/node_modules/**",
        "<node_internals>/**"
      ]
    }
  ],
  "compounds": [
    {
      "name": "Debug: Full Stack",
      "configurations": [
        "Next.js: Server-side: Attach to Process",
        "Next.js: Client-side",
        "API: Express Server"
      ]
    }
  ]
}
```

2. Update your `package.json` development script

It is **IMPORTANT** to update your `package.json` development command to attach the debugger to the underlying Node.js process using `--inspect`, to ensure a smooth debugging flow.
For more details and instructions, check the IDE Debugging: Important Tips section.

3. Start your development server

First, run the server from your terminal: `npm run dev`. You will see a **Debugger listening on...** message in the Console. Make sure the port number that you see matches the `port` in your **attach** configuration.

```
Debugger listening on ws://127.0.0.1:9230/96b1a4cc-ed34-4bf3-a20d-528f67ad0e92
For help, see: https://nodejs.org/en/docs/inspector
   ▲ Next.js 16.0.7 (Turbopack)
   - Local:         http://localhost:3000
   - Network:       http://192.168.2.170:3000
   - Debugger port: 9230
   - Environments: .env.development

 ✓ Starting...
 ✓ Ready in 3.1s
```

> **Note:** Since the debugger is set up with an **attach** configuration, it is essential to run the development server first. Then, only after that, run the debugger, so that the debugger is attached to the underlying development server Node.js process.

4. Start the Debugger in Cursor

After adding this configuration, you can add breakpoints in all layers (e.g., inside `useNoteForm.ts`, `route.ts`, `server.js`) and start the debugger by choosing your compound configuration (`Debug: Full Stack`) and pressing the green **Start Debugging** (▶) icon.
The debugger will now attach to the running Next.js process and will stop in all layers during the debugging flow, allowing you to explore a holistic view of your Next.js application.

Debugging in Cursor: Angular SSR Setup

If an Angular application is created with the Angular CLI, the CLI by default also creates `launch.json` and `tasks.json` configurations in the `.vscode` folder for debugging in Cursor and VS Code IDEs. While these default configurations are suitable for running client-side Angular, to enable a full (client + server) debugging workflow for an SSR-powered Angular application, you need to take extra steps.

Below is a 4-step guide for a full-stack, two-layer debugging setup of an Angular SSR-powered application:

1. Set up your `launch.json` file

This configuration is based on the following steps:

Step A: Create the **`Angular: Client-side`** configuration:

- It is based on the `npm: start` `preLaunchTask` that is specified in the `tasks.json` file.
- The `preLaunchTask` option means that the debugging browser opens with the specified local URL only after the `endsPattern` of that specific task is reached.

Step B: Create the **`Angular: SSR (Server-side): Attach to Process`** configuration:

- It attaches the debugger to the running Node.js process which is the process that is initialized with the `npm run start` command and runs both the Angular SSR server and the client-side.
- It is important to specify the correct port (e.g., `9231`). Since the underlying Angular development Node.js process uses the `9229` inspect port by default, it is advisable to use a custom port to avoid port conflicts. Make sure the same port is specified in `package.json` with `--inspect=9231`.

Step C: Create the **`Debug: Angular SSR`** compound configuration of the two mentioned configurations. This will allow you to run the full debugging flow of the Angular SSR-powered application.

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "ng serve",
      "type": "chrome",
      "request": "launch",
      "preLaunchTask": "npm: start",
      "url": "http://local.angular-shop-app.com:4200"
```

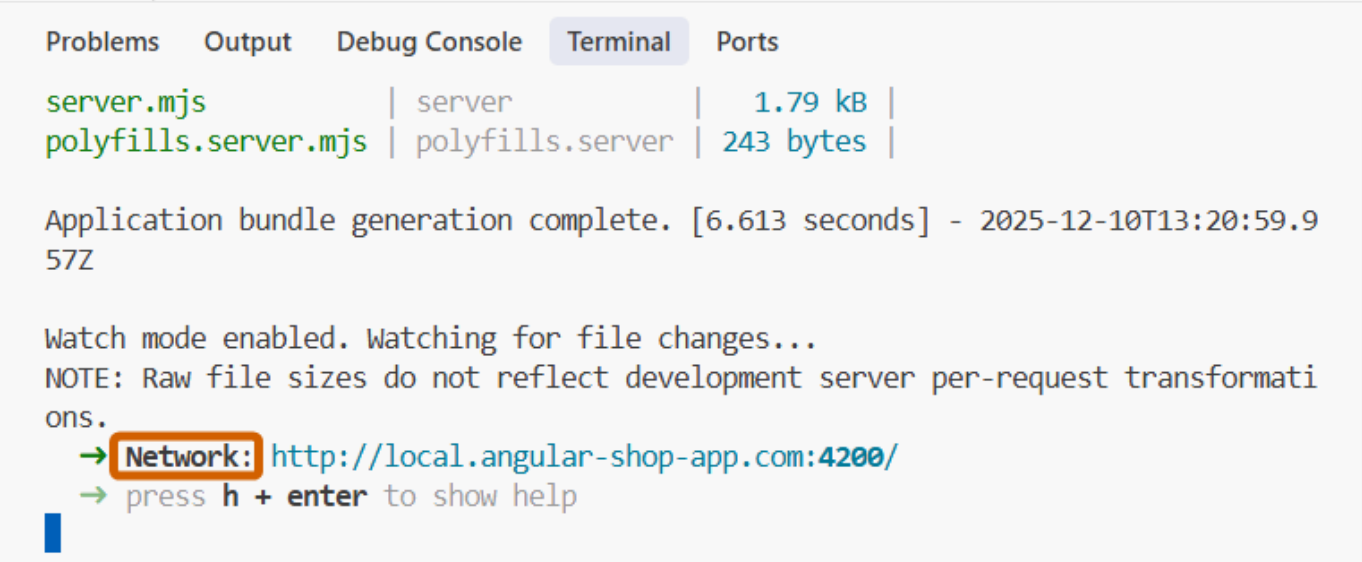
```
    },
    {
      "name": "Angular: Client-side",
      "type": "chrome",
      "request": "launch",
      "url": "http://local.angular-shop-app.com:4200",
      "webRoot": "${workspaceFolder}",
      "sourceMaps": true,
      "preLaunchTask": "npm: start",
      "skipFiles": [
        "<node_internals>/**",
        "**/node_modules/**",
        "**/vite/deps_ssr/**"
      ]
    },
    {
      "name": "Angular: SSR (Server-side): Attach to Process",
      "type": "node",
      "request": "attach",
      "port": 9231,
      "restart": true,
      "localRoot": "${workspaceFolder}",
      "remoteRoot": "${workspaceFolder}",
      "skipFiles": [
        "<node_internals>/**",
        "**/node_modules/**",
        "**/vite/deps_ssr/**"
      ]
    }
  ],
  "compounds": [
    {
      "name": "Debug: Angular SSR",
      "configurations": [
        "Angular: Client-side",
        "Angular: SSR (Server-side): Attach to Process"
      ]
    }
  ]
}
```

2. Set up your `tasks.json` file

Since the client-side configuration (`Angular: Client-side`) in this example is based on the task in `tasks.json`, you need to accurately describe its starting (`beginsPattern`) and ending (`endsPattern`) patterns. This configuration tells the debugger to open the browser only when the development server has started.

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "npm",
      "script": "start",
      "isBackground": true,
      "problemMatcher": {
        "owner": "typescript",
        "pattern": "$tsc",
        "background": {
          "activeOnStart": true,
          "beginsPattern": "(.*?)",
          "endsPattern": "Network:"
        }
      }
    }
  ]
}
```

In this example, the `ng serve` command ended with the word **Network:**, providing the URL for the local development server, so adding it as the `endsPattern` works as expected. However, you need to verify this pattern based on your local setup.

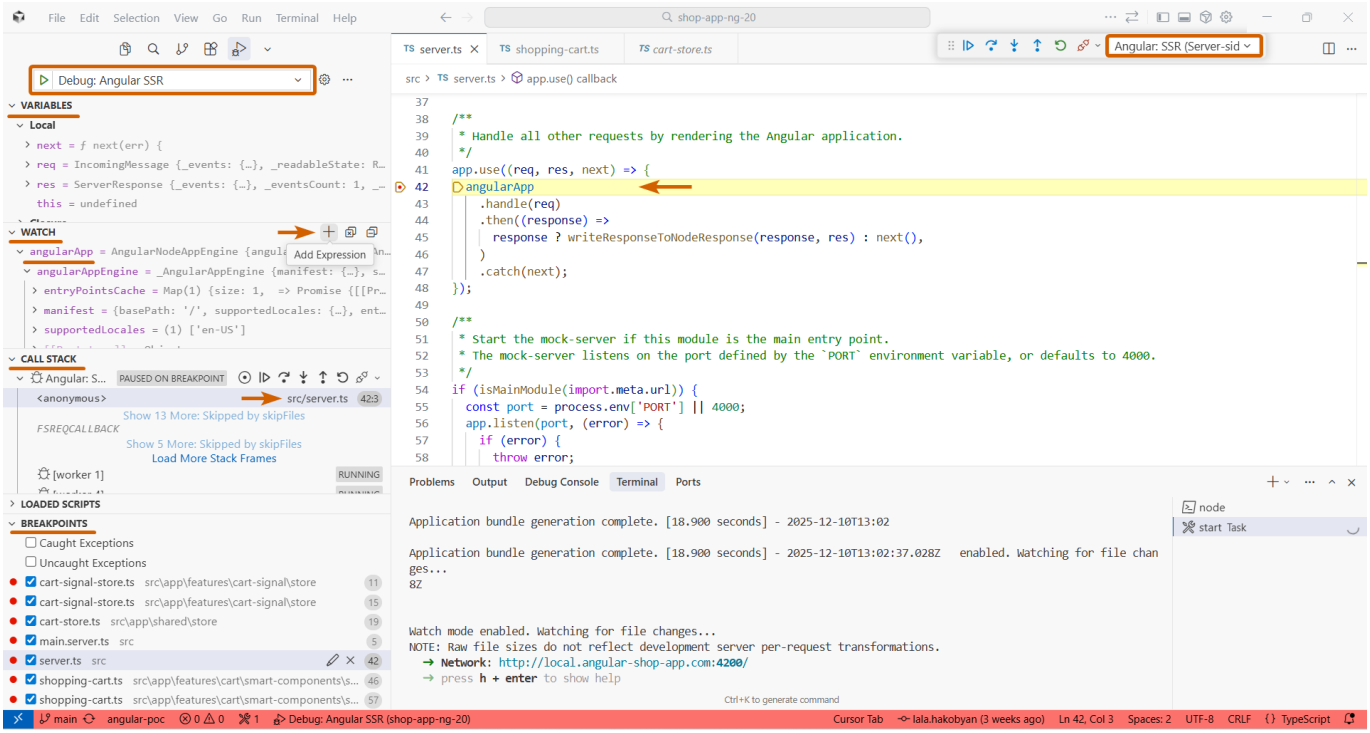


3. Update your `package.json` development script

It is **IMPORTANT** to update your `package.json` development command to attach the debugger to the underlying Node.js process using `--inspect`, to ensure a smooth debugging flow.
For more details and instructions, check the IDE Debugging: Important Tips section.

4. Start the Debugger in Cursor

After adding this configuration, you can add breakpoints in all layers (e.g., inside `app.use` in `server.ts`) and start the debugger by choosing your compound configuration (`Debug: Angular SSR`) and pressing the green **Start Debugging** (▶) icon.
The debugger will run the `npm run start` task, and whenever it is completed, the debugging window will be opened with the corresponding local development URL: `http://local.angular-shop-app.com:4200`. Also, the debugger will be attached to the running Node.js process so you can put breakpoints in the SSR server files as well as on client-side code and debug the full flow.



Debugging in Cursor: Configuration Parameters

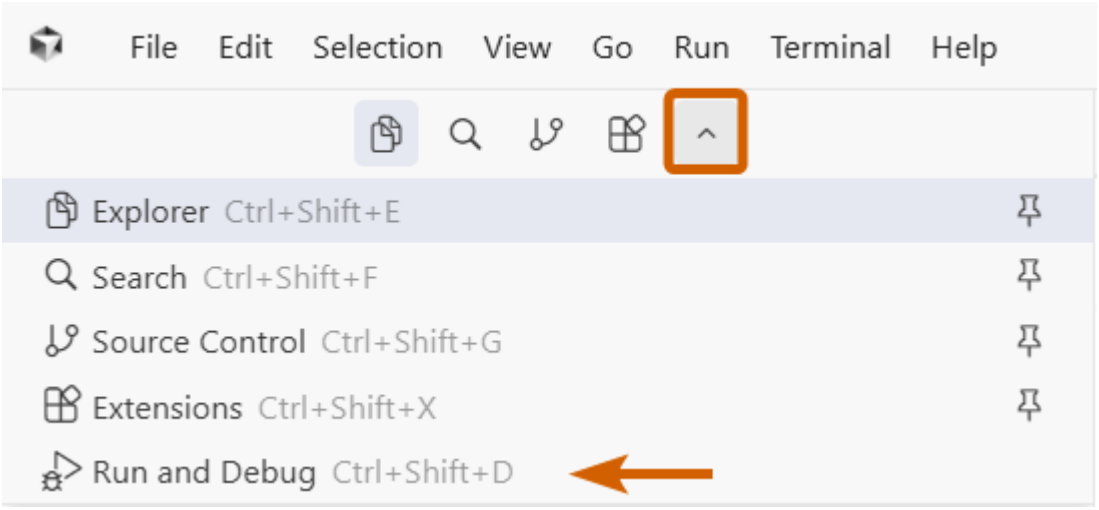
- **`cwd`**: The **current working directory**. This tells the debugger where to run the command from. `${workspaceFolder}` is a variable that points to the root folder of your open project.
- **`command`**: The script from your `package.json` that the debugger should execute (e.g., `npm run server`).
- **`request`**: Can be `"launch"` (starts a new process) or `"attach"` (connects to an existing process).
- **`port`**: For an **attach** request, this specifies the port the debugger is listening on (e.g., `9230`).
- **`url`**: The local URL of your application, used in client-side configurations.
- **`type`**: Specifies the debugger to use (e.g., `node`, `chrome`).
- **`skipFiles`**: An array of file patterns to ignore during debugging.

Debugging in Cursor: Starting Your Debug Session

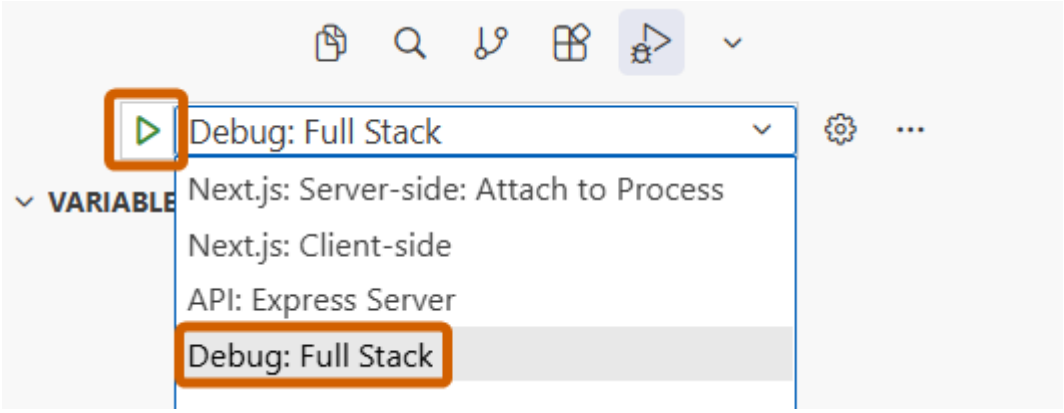
1. Set Breakpoints: In your code, click in the gutter (the area to the left of the line numbers). A **red dot** will appear, indicating a breakpoint. You can set these in your client-side components, server-side code in Next.js, server files in your SSR-based Angular application and Express.js API routes.

2. Launch the Debugger:

- From the dropdown on the top-left of the screen, click on the dropdown icon (**v**) and select the **Run and Debug** option.



- Select your compound configuration (e.g., `Debug: Full Stack` or `Debug: Angular SSR`) from the top dropdown and click the **Start Debugging** (▶) icon.



3. Debug: Cursor will launch all configurations included in your compound setup and open a new browser window. When your application's code hits one of your breakpoints, execution will pause and the **Debugger** window will appear, allowing you to inspect the state and navigate through the code using the debugger controls.

Debugging in Cursor: Navigation Controls and State Inspection

Once the debugger is paused, you can use these tools to control the application execution flow and inspect its state:

Navigation Controls:

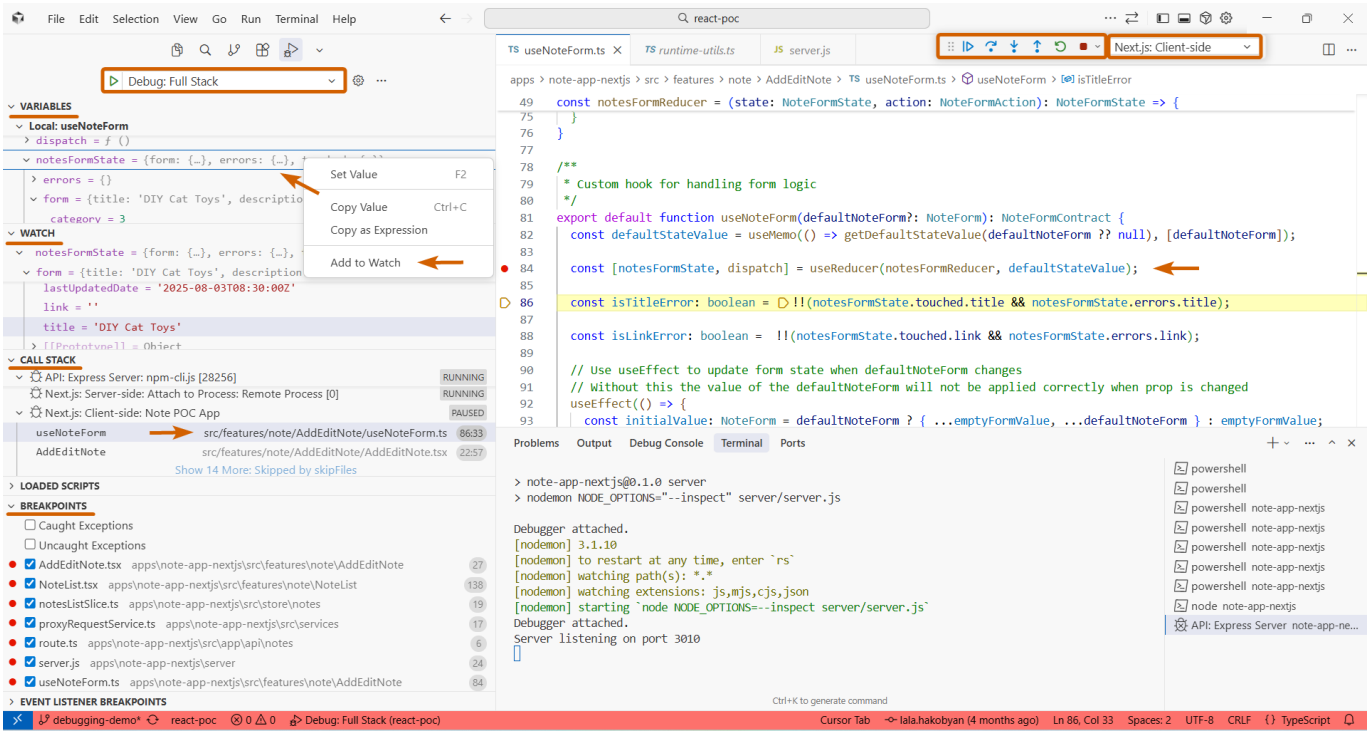
- ▶ **Continue (Windows/Linux/macOS: `F5`)**: Resumes execution until the next breakpoint is hit or the end of the program is reached.
- ↷ **Step Over (Windows/Linux/macOS: `F10`)**: Executes the currently highlighted line and moves to the next one without going inside any functions on that line.

- ↓ **Step Into (Windows/Linux/macOS: `F11`)**: If the current line contains a function, this will move the debugger to the first line inside that function.
- ↑ **Step Out (Windows/Linux/macOS: `Shift+F11`)**: Steps out of the current function and returns to the calling function, to the line right after where it was called.
- ⟲ **Restart (Windows/Linux: `Ctrl+Shift+F5` | macOS: `Cmd+Shift+F5`)**: Restarts the entire debugging session.
- ■ **Stop (Windows/Linux/macOS: `Shift+F5`)**: Ends the debugging session completely.

> **Note for macOS users:** Based on your settings, to trigger the shortcuts involving F-keys, you may need to hold the `Fn` key (e.g., `Fn+F8`).

Inspection Panels:

- **Variables**: See the current values of all variables in scope. Right-click a variable to **Set Value**, **Copy Value** or **Add to Watch**.
- **Watch**: Track the value of specific variables or expressions throughout your debugging session. You can add variables to the **Watch** panel by right-clicking the variable in the **Variables** panel and choosing the **Add to Watch** option. Alternatively, to add a custom expression in the **Watch** panel, hover over the **Watch** panel heading and use the **Add Expression** (+) icon.
- **Call Stack**: View the sequence of function calls that led to the current breakpoint. This acts as a history of your code's execution path.



3.3 IDE Debugging: Important Tips

During my practical debugging process, I found out that passing the `--inspect` flag to the underlying Node.js process that runs behind the development command is essential not only for Cursor debugging, where you attach the debugger to the Node.js process, but also for WebStorm debugging.
WebStorm relies on this to handle the configurations you set up smoothly. Otherwise, you risk the debugger missing breakpoints in some layers. It may stop in one layer (e.g., the server) but skip breakpoints in other layers.

To allow a debugger to attach to the Angular SSR server process (started by `ng serve`) or to the Next.js server process (started by `next dev`), you need to pass the `--inspect` flag to the underlying Node.js process.

- **For macOS and Linux users**: Change your development scripts in `package.json` to:
  - **For Angular**: `"start": "NODE_OPTIONS='--inspect=9231' ng serve --port 4200"`
  - **For Next.js**: `"dev": "NODE_OPTIONS='--inspect' next dev"`
- **For Windows users**: You will need the `cross-env` package for this to work:
  - Install the package: `npm install cross-env --save-dev`
  - Your script will look like this:
    - **For Angular**: `"start": "cross-env NODE_OPTIONS=\"--inspect=9231\" ng serve --port 4200"`
    - **For Next.js**: `"dev": "cross-env NODE_OPTIONS=\"--inspect\" next dev"`

By default, when started with the `--inspect` switch, a `Node.js` process listens for a debugging client (e.g., Chrome DevTools) on `127.0.0.1:9229`. If other Node.js processes are also running in inspect mode and using the same default port, they will conflict with each other.
While some frameworks (e.g., Next.js) may use different inspect ports depending on the setup, Angular CLI usually relies on the default Node.js port.

General Rules to Avoid Port Conflicts

- **Avoid using Node.js default port**
  Run the development server with `--inspect` without specifying a custom port. In the terminal output you will see which underlying port is used. If it uses the default Node.js port `9229`, change it to a custom one, otherwise use the framework-provided inspect port.
  For example, in my setup I changed an Angular SSR application's port to the custom value `9231` because it was using `9229` by default. I didn't change the Next.js port since it was using `9230` as default and worked without conflicts on my local machine.
- **Choose unique port number and clean up stale port processes**
  - If you are running different projects and debugging them at the same time, consider providing custom unique inspect ports for each setup.
  - Make sure to clean up running processes bound to the port before you start debugging.

These simple steps will help you avoid port conflicts and ensure a smooth debugging session.

3.4 IDE Debugging: Troubleshooting Steps

If your breakpoints are unbound during debugging, or the debugger goes into an endless loading mode, below are proven troubleshooting steps that will help you restore your debugging flow.

1. Check if Ports are Free

Sometimes ports are being occupied by old processes or multiple processes are running on the same port. Check local development ports like `4200` or Node.js inspector port like `9231` to make sure they are free for usage.
This is the most probable reason of your unbound breakpoints, so make sure to check it first.

To find and kill the processes using a specific port (e.g., `4200`), execute these commands in **PowerShell** or **Command Prompt** in **Windows** and in **Terminal** for **macOS / Linux**:

- Find all processes using port `4200`:
  - **Windows:** `netstat -ano | findstr :4200`
  - **macOS / Linux:** `lsof -i :4200`
- It will return results like these:
  - **Windows:** `TCP 0.0.0.0:4200 0.0.0.0:0 LISTENING 12345`
  - **macOS / Linux:** `node 12345 youruser 0u IPv4 0x854013108bce2194 0t0 TCP *:4200 (LISTEN)`
- The number `12345` in this example is the PID. You need to kill all found PIDs using this command:
  - **Windows:** `taskkill /PID 12345 /F`
  - **macOS / Linux:** `kill -9 12345`

2. Restart the built-in IDE debugger engine

IDEs sometimes hold stale debugging sessions which may cause inconsistent debugging flow. Below are steps to restart your current debugging session:

- Stop debugging by clicking the corresponding stop icon (■).
- Close all terminals related to the debug session, including the opened browser window.
- Choose your debug configuration one more time and start the debugging session again.

3. Invalidate IDE Caches / Restart

IDEs may also store corrupted cache from the previous state of the application that can affect your debugging flow. To clean the IDE cache, follow these steps:

- For WebStorm, use the official action for corrupted caches:
  `File > Invalidate Caches… > Invalidate and Restart`
- For Cursor, there is no **Invalidate Caches** button, but reloading the window clears the internal state:
  - Press `Ctrl+Shift+P` for Windows/Linux or `Cmd+Shift+P` for macOS
  - Run: `Developer: Reload Window`

4. Clean Up Project Metadata

In some cases, your project metadata stored in the IDE can be outdated and cause an inconsistent debugging flow. To reset all project-specific or IDE-specific workspace metadata, follow the instructions below:

Step A: Close the IDE.

Step B: Delete the appropriate metadata folder:

- **WebStorm**: Delete the project's `.idea` folder.

  IMPORTANT: Before deleting the `.idea` folder, make sure to save the `runConfigurations` folder that stores debug configurations or any other important information, otherwise you risk losing them.

- **Cursor**: Delete the IDE cache folders in the file system.

  Go to the `workspaceStorage` folder. The projects here will be marked by MD5-hash-named folders.

  ```
  fe08dfbe5a137b32715a5b9e31175c13
  d031b326177360acb27a0379c62df64c
  ```

  You can find the specific projects by date. Once found, you have two options:

  - Remove the `ms-vscode.js-debug` folder inside of it to clean up debugging metadata.
  - If it doesn't fix the issue, you can remove the project folder or the whole `workspaceStorage` entirely.
    IMPORTANT: Deleting `workspaceStorage` or a project folder inside of it also **deletes your chat history**. The `workspaceStorage` folder contains SQLite databases (`state.vscdb` files) that store workspace state and chat history. So, make sure to back up the folders before removing them if you care about chat history.

  Cursor `workspaceStorage` location per OS:

  - **Windows**: `C:/Users/<Username>/AppData/Roaming/Cursor/User/workspaceStorage`
  - **macOS**: `/Users/<Username>/Library/Application Support/Cursor/User/workspaceStorage`
  - **Linux**: `/home/<Username>/.config/Cursor/User/workspaceStorage`

Step C: Reopen the project (IDE regenerates its metadata automatically).

> **Note:** The path of the Cursor `workspaceStorage` folder may vary based on the OS version or Cursor version, so please check its current path in your installed Cursor folder.
> **Be careful not to remove anything else** besides the mentioned folders. Otherwise, you risk corrupting the currently installed Cursor and may lose some of your setup or history.

3.5 IDE Debugging: Common Use Cases

IDE debugging is an essential tool for multi-layer applications like a **Next.js application** or an **SSR-powered Angular application**, which have separate front-end, server-side and API layers.

- **Get a holistic understanding of your complex, multi-layer front-end application:** IDE debugging allows you to seamlessly trace the execution flow across these different layers and get a complete and holistic view of your application's execution logic, which can be difficult, confusing and time-consuming if using only `console.log`.
- **Inspect application state in real time:** With an IDE debugger, you can pause your code at any point, inspect the call stack and watch variables change in real time, getting a complete picture of your application's state.
- **Understand how framework-specific features work:** With the IDE debugger attached to all layers, you can set breakpoints in different layers (client-side, server-side and API) which will help you understand how framework-specific features work, like Angular SSR and hydration flow or Next.js server and client components.
- **Debug external libraries/tools:** In some cases, you may need to step into and debug external libraries/tools maintained by separate providers or other teams in your organization. The IDE debugger is a convenient tool for that workflow because you would easily get lost in your `console.log` entries if you try to add them in a codebase you are not familiar with.

4. Framework and Library Specific Debugging Tools

Besides the general-purpose developer tools, many frameworks and libraries offer their own specialized extensions and APIs. These tools provide a deeper dive into framework-specific concepts, making it much easier to debug complex applications.

In this chapter, you will explore the following debugging tools:

- React DevTools Extension
- React Profiler API (Programmatic Profiling)
- Angular DevTools
- Redux DevTools

4.1 React DevTools

React DevTools is an essential browser extension that lets you inspect the React component hierarchy of your application, view and edit component state and props and profile performance to identify bottlenecks.

It's a free browser extension available for most major browsers.

- **Chrome**: Chrome Web Store
- **Firefox**: Firefox Add-ons
- **Edge**: Microsoft Edge Add-ons

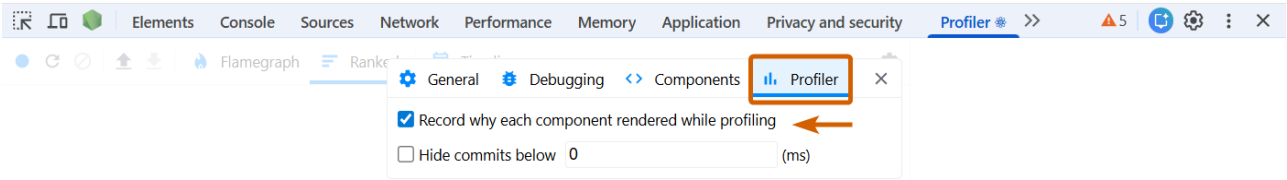
Once installed, you'll see three new panels in your browser's DevTools: **Components**, **Profiler** and **Suspense** (new feature, available in recent versions of React DevTools).

React DevTools: Useful Tips Before Starting Debugging

Before you start debugging, enable a couple of settings for a deeper analysis. Go to any React DevTools panel and click on the **Settings** (⚙) icon:

1. Under the **Profiler** tab:

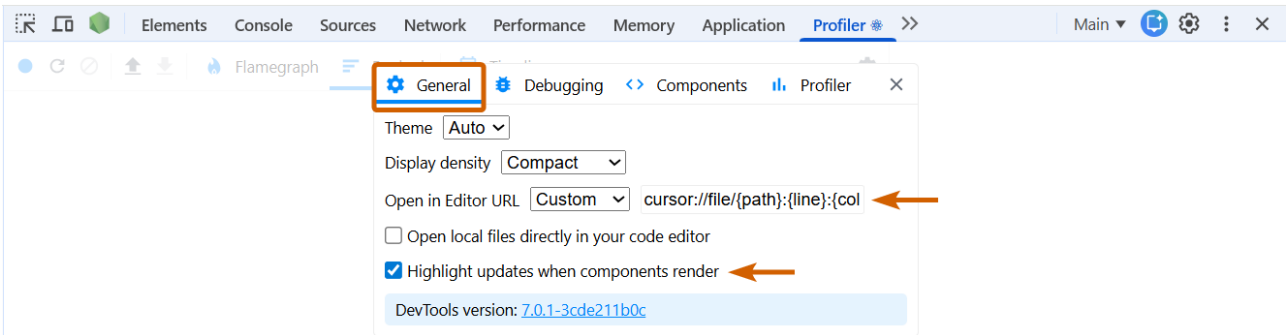
- Check the **Record why each component rendered while profiling** setting. This is an important setting used to identify the exact cause of a re-render.



2. Under the **General** tab:

- Choose your IDE name and specify the URL pattern for your source files in the **Open in Editor URL** setting so that you can open your source files from the browser.

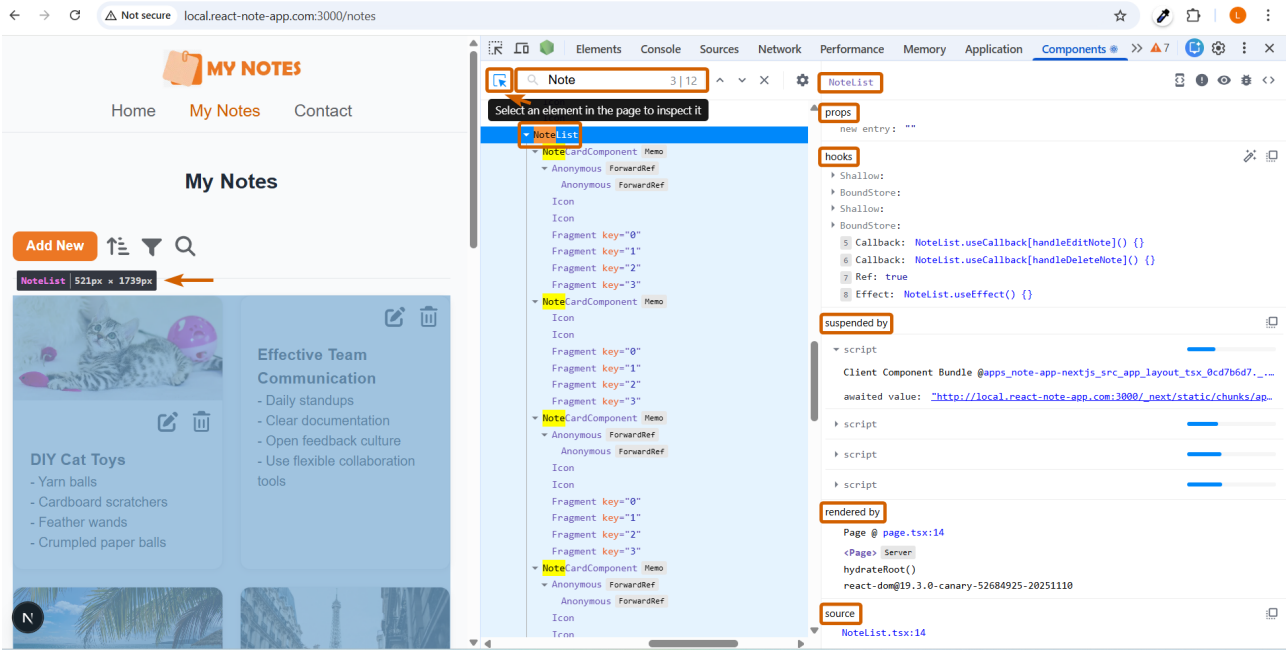
- Check the **Highlight updates when components render** option. This setting enables a visual overlay that briefly highlights parts of the page when their React components re-render. Use it to quickly spot which UI areas update due to state, props, context changes or parent re-renders.
  Note: Highlighting React renders may be annoying sometimes. You may consider activating it during local development for finding re-render bottlenecks and deactivating it otherwise.



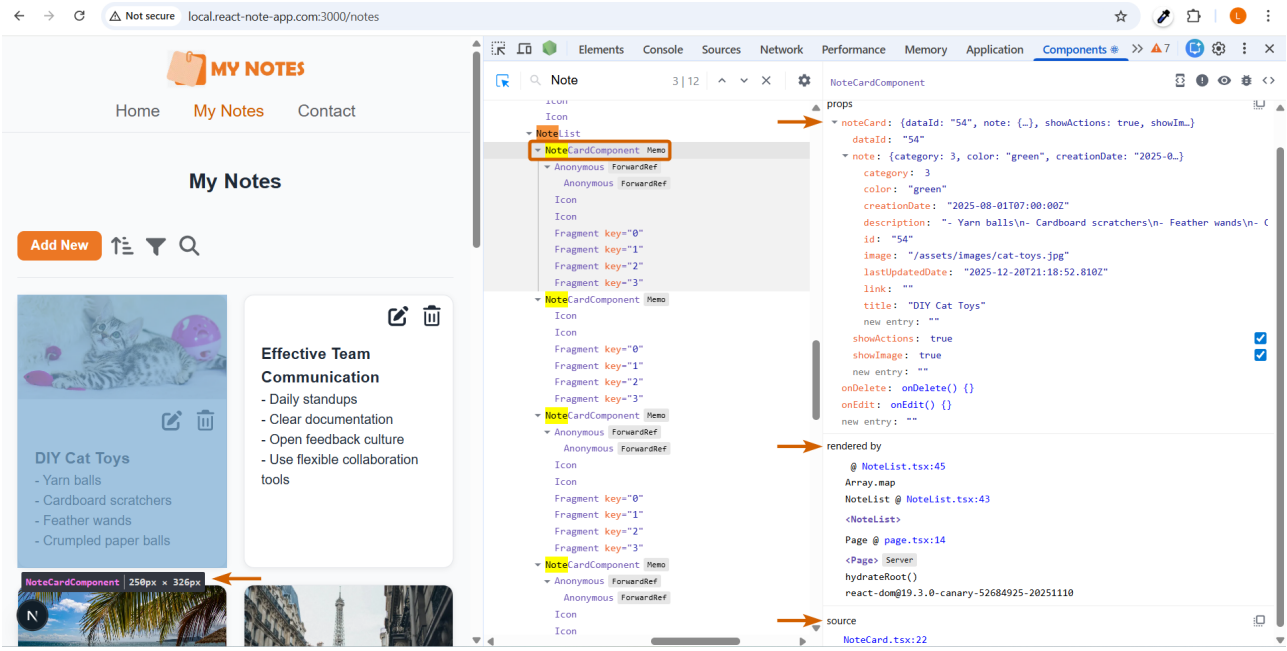
React DevTools: Components Panel

The **Components** panel is the primary tool for inspecting your React application's structure and state. You can think of this as the **Elements** panel, but for your React components instead of the DOM.

- **Component Tree (Left Pane)**: Shows the hierarchical structure of your rendered React components, helping you understand how components are nested and which component renders which.
    - The **search bar** at the top lets you search for components by name or pattern.
    - The **arrow** at the top-left lets you select an element in the page to inspect it.



- **Details Pane (Right Pane)**: When you select a component from the tree, this pane shows you everything you need to know about it:
    - **Props**: View all the props passed to the selected component.
    - **Hooks**: Inspect the values of all hooks (e.g., `useState`, `useContext`).
    - **Suspended by**: Identify what caused this component to suspend (pause rendering), for example, a lazy-loaded bundle or Suspense-based data source. If a `Suspense` boundary is involved, you can click it to view its props such as `children` or `fallback`.
    - **Rendered by**: Inspect the chain of components that led to the selected component being rendered. You can click on each component from the chain to navigate to that component and view its details.



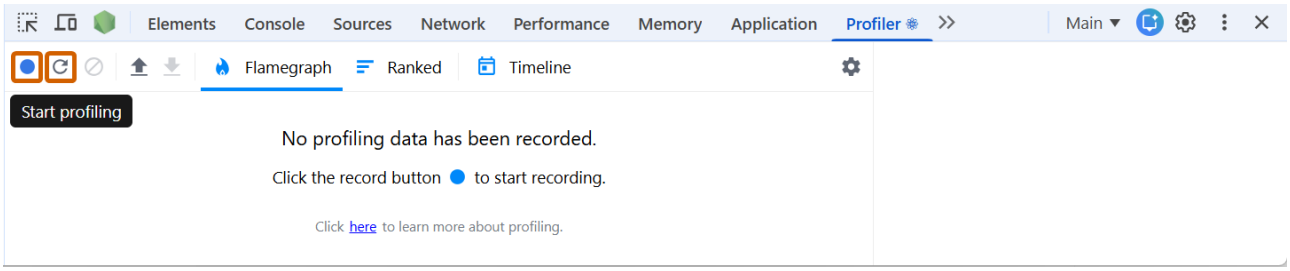
React DevTools: Profiler Panel

The **Profiler** panel is a useful performance analysis tool that helps you find and fix React-specific performance issues, like unnecessary re-renders.

Profiler Panel: How to Start Debugging

Follow these steps to start debugging:

1. Open the **Profiler** panel in React DevTools.
2. Click the blue circle icon (🔵) to **start profiling** or the reload icon (⟳) to **reload and start profiling**.



3. Interact with your web page by performing actions in your application (e.g., clicking a button, editing, deleting).
4. Click the red circle icon (🔴) to **stop profiling**. The profiler will then present a detailed report of the captured data.

Profiler Panel: Understand Performance Data

To understand the performance data displayed on the React DevTools Profiler, you need to dive deeper into what rendering actually means in React.
Conceptually, React does render work in three steps: **Trigger** → **Render** → **Commit**:

1. Trigger a render

There are two reasons for a component to render:

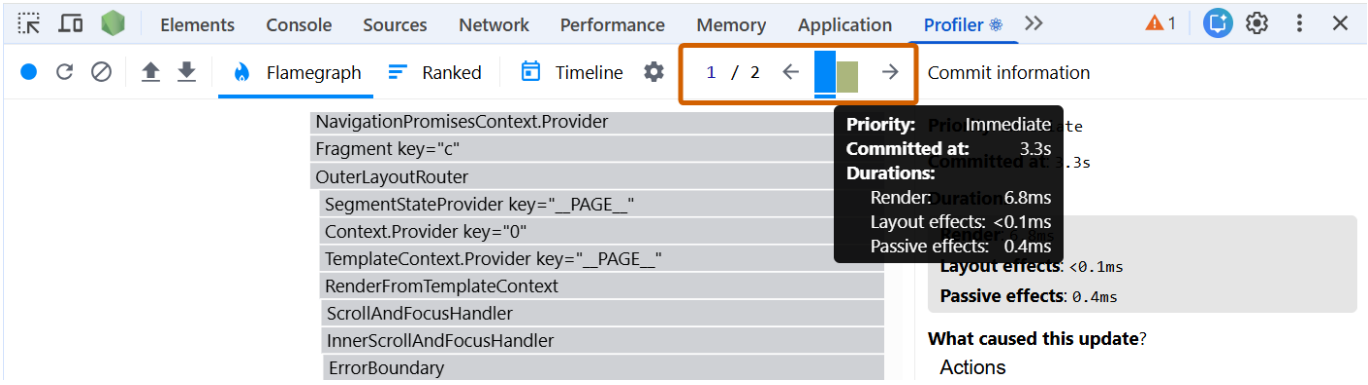
- It's the component's initial render.
- The component's (or one of its ancestors') state has been updated.

2. React renders your components
After you trigger a render, React calls your components to understand what was changed and to figure out what to display on screen. **Rendering** is the process of **React calling your components**.

3. React commits changes to the DOM
After rendering (calling) your components, React will modify the DOM: this is when React inserts, updates, and removes DOM nodes. React only changes the DOM nodes if there's a difference between renders. This phase is called **Commit**. React also calls lifecycles like `componentDidMount` and `componentDidUpdate` during this phase.

- If you define the `componentDidMount()` method, React will call it when your component is added (mounted) to the screen.
- If you define the `componentDidUpdate()` method, React will call it immediately after your component has been re-rendered with updated props or state. This method is not called for the initial render.

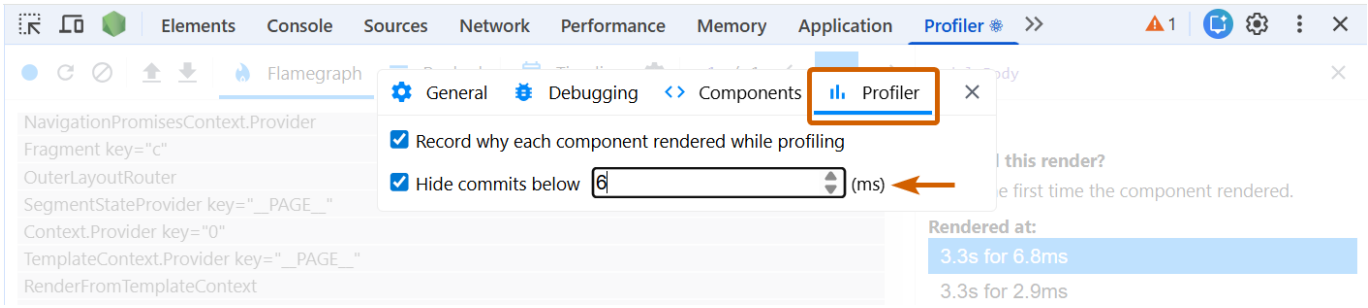
The React DevTools profiler groups performance info by **commit**. Commits are displayed in a bar chart at the top right of the **Profiler** panel:



Each bar in the chart represents a single commit with the currently selected commit underlined and colored in a specific color (in this example, `blue`). You can click on a bar (or the left/right arrow buttons) to select a different commit.
The color and height of each bar correspond to how long that commit took to render: the taller a bar is, the longer it took to execute.

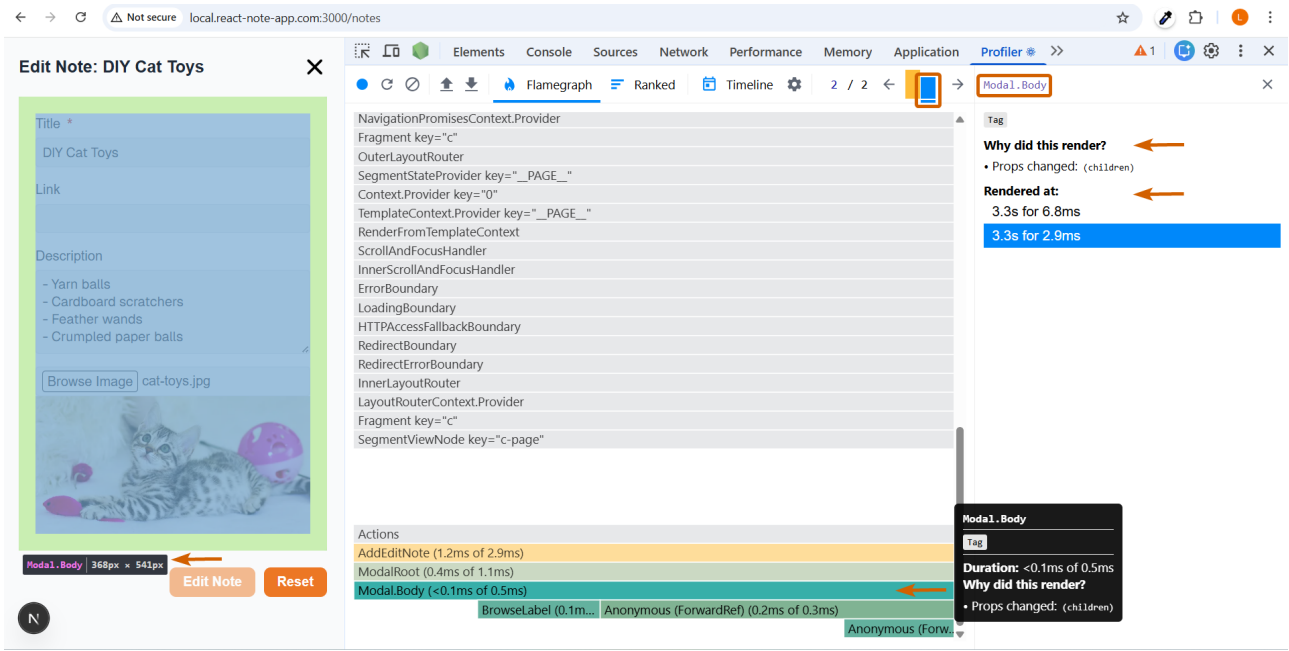
Profiler Panel: Filter

In some cases, you may record a longer profile and end up with too many commits which will be hard to process. The profiler offers a filtering mechanism to help with this. In **Settings > Profiler**, use the **Hide commits below** setting to specify a threshold. The profiler will then hide any commits faster than that value.



Profiler Panel: Main Tabs

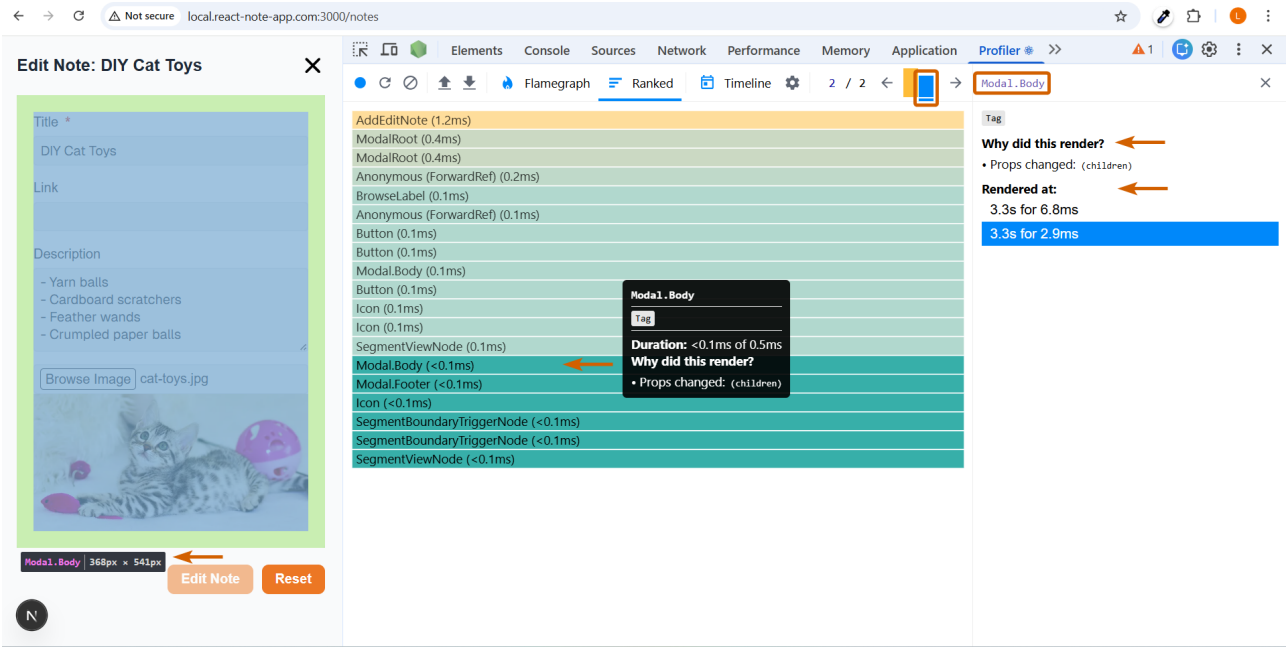
- **Flamegraph Chart**: Represents the state of your application for a particular commit expressed in the component tree hierarchy (parent-child relationship). Each bar in the chart represents a React component (e.g., `Actions`, `AddEditNode`) where child components appear below their parents. Wider components are good candidates for optimization because the wider a component is, the longer it took to render.



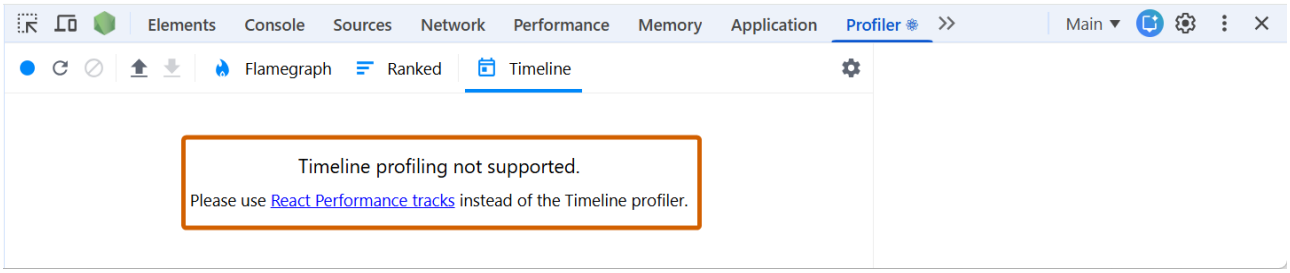
  - The width of a bar indicates how long it took to render the component (and its children) when they last rendered. If the component did not re-render as part of this commit, the time represents a previous render.
  - The color of a bar indicates how long the component (and its children) took to render in the selected commit.

  In this example, the yellow component took more time, green components took less time and gray components did not render at all during this commit.

- **Ranked Chart**: Represents the state of your application for a particular commit but with a simpler view that orders all rendered components by their duration, with the components that took the longest to render showing at the top. The Ranked Chart is often faster for identifying the **most expensive** component, whereas the Flamegraph is better for understanding parent-child relationships.



- **Timeline**: In earlier versions, this tab showed component renders over time, so you could see when each render happened and if multiple renders overlapped. This helped to identify rendering patterns and performance bottlenecks.
  However, according to the latest versions of React DevTools, the React team advises using React-specific Custom Performance tracks within the Performance panel trace. This aligns with a recent industry shift where frameworks provide a unified debugging experience, allowing developers to review application performance from a single location: the browser **DevTools Performance panel**.



  To access this unified view with **React Performance tracks**, switch from the **React DevTools** panel to the main **Performance** panel in your browser DevTools and record a new trace.
  You can learn more about it in the upcoming React Custom Performance Tracks section.

React DevTools: Suspense Panel

Suspense is a new panel available in recent React DevTools versions. It was introduced with the release of React 19. If you don't see it, you may need to update your React DevTools extension to the latest version.

Suspense Compatibility and Limitations

To debug Suspense resources, you need to understand how it works and when it is activated.

Suspense is a React feature that lets you display a fallback until its children have finished loading.

Example: Display `<AdBannerLoader>` fallback until `<AdBanner>` is loaded:

```
// Display `AdBannerLoader` fallback until `AdBanner` is loaded
<Suspense fallback={<AdBannerLoader />}>
  <AdBanner />
</Suspense>
```

- Only Suspense-enabled data sources will activate the `Suspense` component. They include:
  - Data fetching with Suspense-enabled frameworks like **Relay** and **Next.js**
  - Lazy-loading component code with `lazy`
  - Reading the value of a cached Promise with `use`
- `Suspense` does not detect when data is fetched inside an `Effect` or event handler.
- Suspense-enabled data fetching without the use of an opinionated framework is not yet supported.

Suspense Debugging

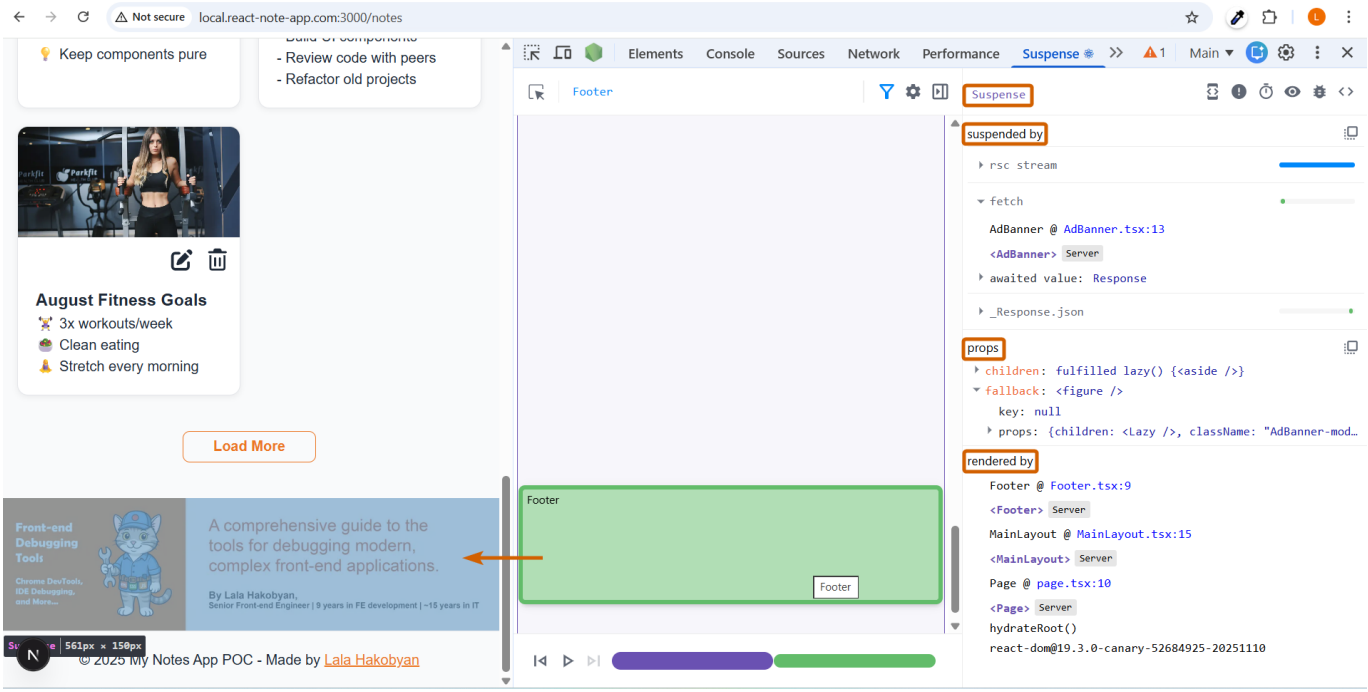
The **Suspense** panel allows you to view **Suspense-based resources** and **Suspense boundary resources** as a list of bars. You can click on each bar to see the highlighted element on the screen.

Additionally, you can review the following useful properties:

- **Suspended by**: Shows what caused this component to suspend (pause rendering), for example, a lazy-loaded bundle or a `fetch` request.
- **Props**: Shows additional details such as `children` or `fallback` if a ***Suspense boundary is explicitly involved****.
  - **children**: The actual UI you intend to render. If `children` suspends while rendering, the `Suspense` boundary will switch to rendering the fallback.
  - **fallback**: An alternate UI to render in place of the actual UI while it is being loaded (e.g., fallback skeleton image, loading spinner, loading text).
- **Rendered by**: Shows the parent component where the resource was rendered and a chain of its parent components (ancestors).

Basically, this is similar to the information you can see in the **Components** tab for Suspense-based resources and boundaries. The main advantage is that it allows you to review all Suspense-based data sources and boundaries as a flat list in a dedicated section for quicker and easier access.

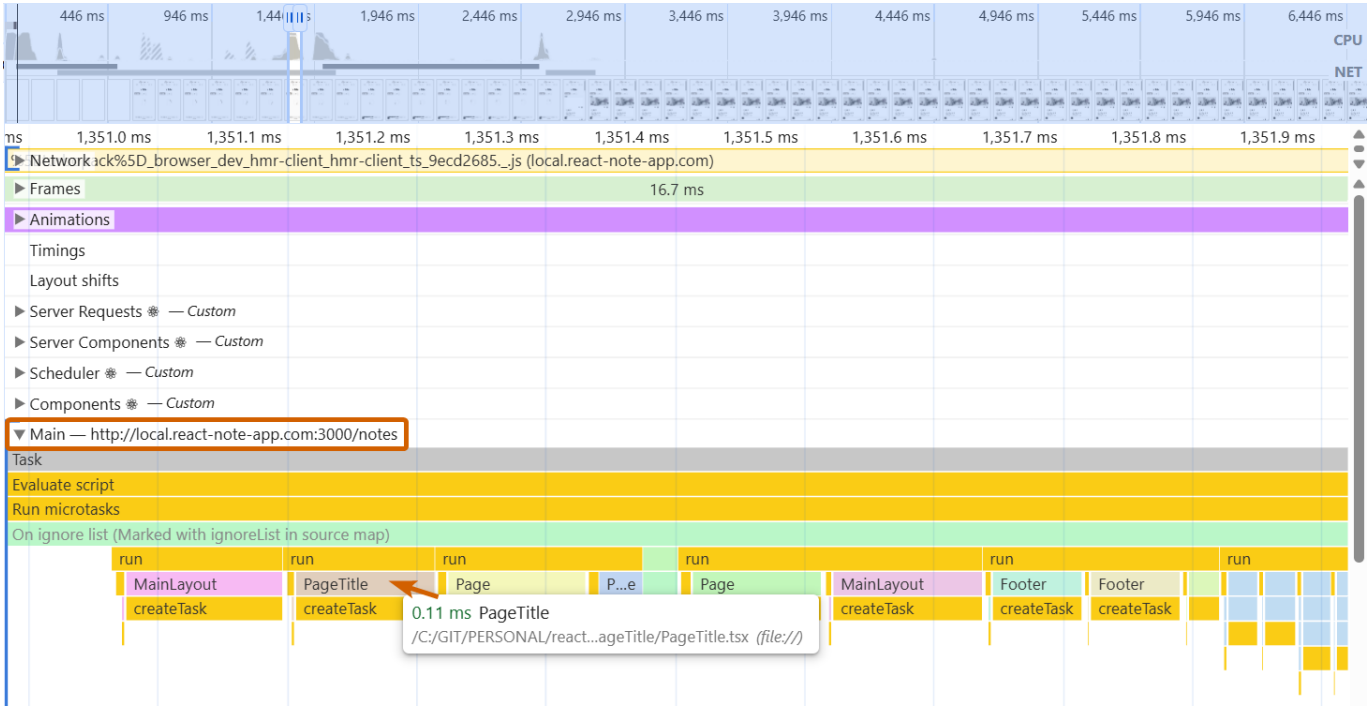
> **Note:** In this context, **Suspense boundary is explicitly involved** means that you explicitly use `Suspense` in your framework-based React code (e.g., Next.js). In contrast to this, React-based frameworks may use Suspense under the hood (e.g., Next.js within the App Router architecture).



React Custom Performance Tracks

Let's explore the data that React allows you to view in the **Performance** panel trace within **React Custom Performance tracks**. We will review examples using the Chrome browser, but you can use any Chromium-based browser (e.g., Microsoft Edge).

After you record a performance trace in the **Performance** panel of your browser DevTools, without custom performance tracks, all you can view is the JavaScript work timing associated with each component (e.g., `PageTitle`, `MainLayout`). This is what the Performance panel extracts by default from your React application. However, this doesn't provide a deep overview of the React working process and specific events that could lead to performance bottlenecks.



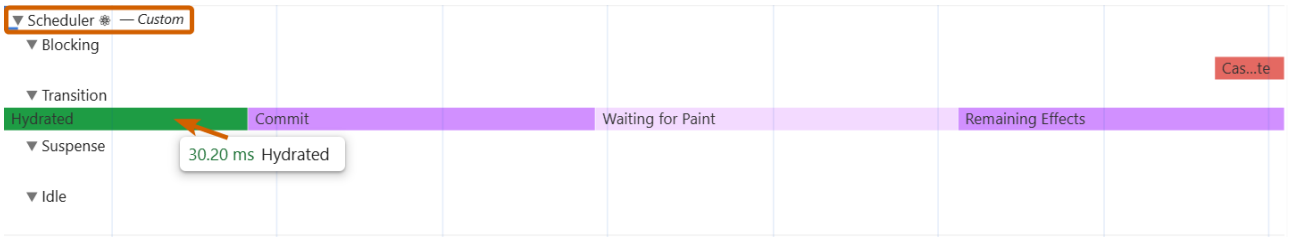
React Performance tracks fill this gap by adding specialized custom entries that appear on the Performance panel's timeline in your browser **DevTools**.
These tracks are designed to provide developers with comprehensive insights into their React application's performance by **visualizing React-specific events and metrics alongside other critical data sources**, such as network requests, JavaScript execution and event loop activity. All of this data is synchronized in a unified timeline within the Performance panel for a complete understanding of application behavior.

After recording a performance trace on the **Performance** panel of your browser DevTools, you will see the following custom tracks to debug the performance of your React application:

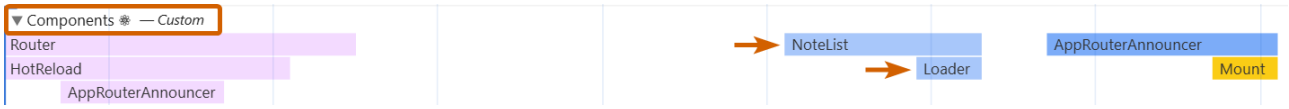
- **Scheduler**
  The Scheduler is an internal React concept used for managing tasks with different priorities. This track consists of 4 subtracks, each representing work of a specific priority:

  - **Blocking:** Synchronous updates, which could have been initiated by user interactions.
  - **Transition:** Non-blocking work that happens in the background, usually initiated via `startTransition`.
  - **Suspense:** Work related to Suspense boundaries, such as displaying fallbacks or revealing content.
  - **Idle:** The lowest priority work that is done when there are no other tasks with higher priority.

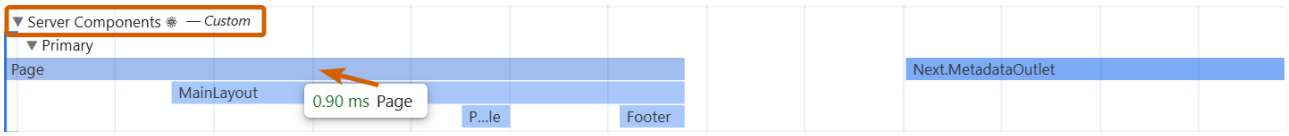
  Every render pass consists of multiple phases that you can see on a timeline: **Update**, **Render**, **Commit** and **Remaining Effects**.



- **Components**
  The Components track visualizes the durations of React components. The data is displayed as a flamegraph, where each entry represents the duration of the corresponding component render and all its descendant components (e.g., children, grandchildren, etc.). It also displays additional details during the render and effects phase, such as **Mount** and **Unmount**.



- **Server Components**
  The Server Components track visualizes the duration of promises awaited by React Server Components during their rendering process. Timings are displayed as a flamegraph, where each entry represents the duration of the corresponding component render and all its descendant components (e.g., children, grandchildren, etc.).



- **Server Requests**
  The Server Requests track visualizes all promises that are used within React Server Components. This includes any async operations like calling `fetch` or async Node.js file operations.
  For example, the Suspense banner rendered on the server side will show in the **Server Requests** track because it is executed asynchronously and streamed within a React Server Component inside a Next.js project.
  Note: Requests that finish before the browser connects, such as blocking initial renders without Suspense, may not appear in the trace.

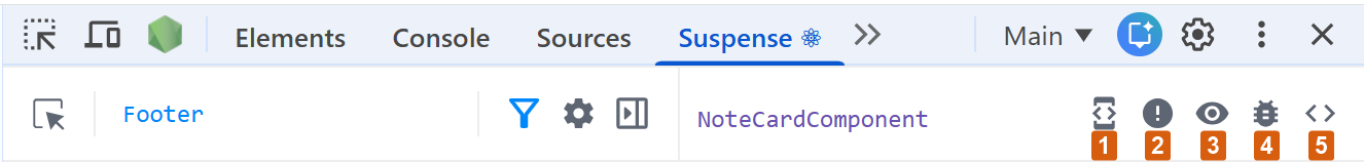
React Custom Performance Tracks: Useful Debugging Tips

- To view custom tracks in the performance trace, you need to enable the **Show custom tracks** checkbox in the **Capture settings** toolbar at the top of the Performance panel.
- React Performance tracks are only available in **development** and **profiling** builds of React.
- **Server Components** and **Server Requests** tracks only appear when debugging applications with server-side rendering, particularly **React Server Components (RSC)** (e.g., in frameworks like Next.js).
- **Filter your performance trace to exclude third-party scripts** from browser extensions to get more accurate results. For example, exclude entries from `installHook.js`, as it is an internal script from the **React DevTools** extension that adds unnecessary noise to your trace.

> **Note: React Custom Performance tracks** appear in Chromium-based browsers (e.g., Google Chrome, Microsoft Edge) in the DevTools **Performance** panel and are separate from the **React DevTools** extension.

React DevTools: Useful Actions

The following useful actions are available in the **Components**, **Profiler** and **Suspense** panels to assist you during your debugging process. You can view them at the top-right of the **Details Pane** (Right Pane), after selecting a specific component.



1. Open the source file in your IDE.
This is controlled by the **General > Open in Editor URL** setting mentioned earlier in the React DevTools:

Useful Tips Before Starting Debugging section.
2. Force the selected component into an errored state.
3. Inspect the matching DOM element.
4. Log the selected component data to the **Console**.
5. View the source of the selected component in the browser **Sources** panel.

React DevTools: Common Use Cases

- **Debug React-based performance issues** such as finding unnecessary re-renders, expensive long-running calculations or the root cause of slow interactions, using the **Profiler panel**. The best scenario to debug these kinds of issues is the following:
    - Start profiling by clicking the blue circle icon (🔵).
    - Interact with your web page by performing actions in your application that cause the slowness (e.g., click the edit modal icon).
    - Stop the recording and select the **Ranked chart** to see which operations took the most time.
    - Click on those high-cost elements and check how many times your app re-rendered for the specific action across all commits.

  If you see a high **commit count** for a simple action or a commit longer than `16ms` (frame max budget), these are signs of a problem.
  Note: Even if you see some re-renders for a single action, they are not a concern if they don't cause slowness. Otherwise, you can use `useMemo`, `useCallback` or `React.memo` for older React versions, or integrate the **React Compiler** (for React 17+) to handle these optimizations automatically.

- **View the overall hierarchy of your components and inspect detailed props and source information**, using the **Components** panel.

- **Debug lazy-loaded components and their fallback states** (`skeleton`, `loader` or `fallback image`), using the **Suspense** panel.

- **Emulate different scenarios for more convenient debugging**: Trigger Suspense fallback, force the selected component into an error state or log the component in the **Console**, using the **interactive actions** in the **Components** and **Suspense** panels.

- **Get a complete understanding of your React application's performance** by visualizing React-specific events and metrics alongside other critical data sources such as network requests, JavaScript execution and event loop activity using **React Performance tracks** in the **Performance** panel.

4.2 React Profiler API (Programmatic Profiling)

The React Profiler API is a tool for measuring the rendering performance of React components directly within your application code. It helps identify performance bottlenecks and areas that could benefit from optimization, such as memoization.

React Profiler API: When It Fires

The Profiler is designed to measure the **commit** phase of a render, which happens when React applies changes to the DOM. This occurs in two situations:

- **Mount (Initial Render):** The `onRender` callback fires the very first time the components inside the `<Profiler>` are rendered. The `phase` argument in the callback will be **`mount`**.
- **Update (Re-render):** The `onRender` callback fires on every subsequent re-render triggered by a change in state, props or context. The `phase` argument will be **`update`**.

React Profiler API: How to Integrate

To use the React Profiler API, wrap your component inside `<Profiler>` and provide an `onRender` callback to process the metrics. React calls the `onRender` callback every time components update within the profiled tree.

Parameters

- **id:** The string id prop of the `<Profiler>` tree that has just committed. This lets you identify which part of the tree was committed if you are using multiple profilers.
- **phase:** This lets you know whether the tree has just been mounted for the first time or re-rendered due to a change in props, state or context. The values are: `mount`, `update` or `nested-update`.
- **actualDuration:** The number of milliseconds spent rendering the `<Profiler>` and its descendants for the current update.
  This indicates how well the subtree makes use of memoization (e.g., `React.memo` and `useMemo`).
  Ideally this value should decrease significantly after the initial mount as many of the descendants will only need to re-render if their specific props change.
- **baseDuration:** The number of milliseconds estimating how much time it would take to re-render the entire `<Profiler>` subtree **without any optimizations**.
  You can compare `actualDuration` against `baseDuration` to see if memoization is working.
- **startTime:** A numeric timestamp for when React began rendering the current update.
- **commitTime:** A numeric timestamp for when React committed the current update.
  This value is shared between all profilers in a commit, enabling them to be grouped if desirable.

Example: Log any renders that are slower than the 16ms frame budget

```
import { Profiler } from 'react';

type ProfilerPhase = 'mount' | 'update' | 'nested-update';

function onRender(
    id: string,
    phase: ProfilerPhase,
    actualDuration: number,
    baseDuration: number,
    startTime: number,
    commitTime: number
): void {
    const slowThreshold = 16;
    if (actualDuration > slowThreshold) {
        console.warn(
            `[Profiler][${id}] ${phase} render took ${actualDuration.toFixed(2)}ms`
        );
    }
}

export default function Actions() {
    return (
        <Profiler id="ActionsComponent" onRender={onRender}>
            <div className={styles.actions}>
                {/* Your component's content goes here */}
            </div>
        </Profiler>
    );
}
```

React Profiler API: Common Use Cases

- **Performance Optimizations:** By wrapping specific component subtrees with the `Profiler`, you can isolate and measure their rendering costs.
    - Wrap specific components in `<Profiler>` and identify components that are re-rendering unnecessarily or contributing most to slow updates (bottlenecks).
    - Perform optimizations by manually applying `React.memo`, `useCallback`, `useMemo` or integrating **React Compiler** for automatic optimizations.
    - After that, use the `<Profiler>` again to verify the effectiveness of those updates by comparing the render times before and after your changes or by comparing `actualDuration` with `baseDuration`.

- **Production Performance Monitoring:** The `onRender` callback provides detailed timing data (`actualDuration`, `baseDuration`, etc.). With a profiling-enabled build, this data can be captured from real users and sent to a performance monitoring platform (e.g., **Datadog**, **Sentry**) to continuously track performance and identify regressions over time.
- **Understanding Component Lifecycle and Interactions:** The `Profiler` provides insights into when components `mount` and `update`. This offers a deeper understanding of the component lifecycle and helps visualize how user interactions or data changes affect the application's rendering performance.

> **Important Note:** Profiling adds some overhead, so it is disabled in React's standard production build. To use it in production, you must opt in by building your application with a special profiling-enabled build.

4.3 Angular DevTools

Angular DevTools is a browser extension that provides debugging and profiling capabilities for Angular applications. It lets you inspect the component structure and profile the application's performance by visualizing its change detection cycles.

It's a free browser extension available for Chrome and Firefox-based browsers.

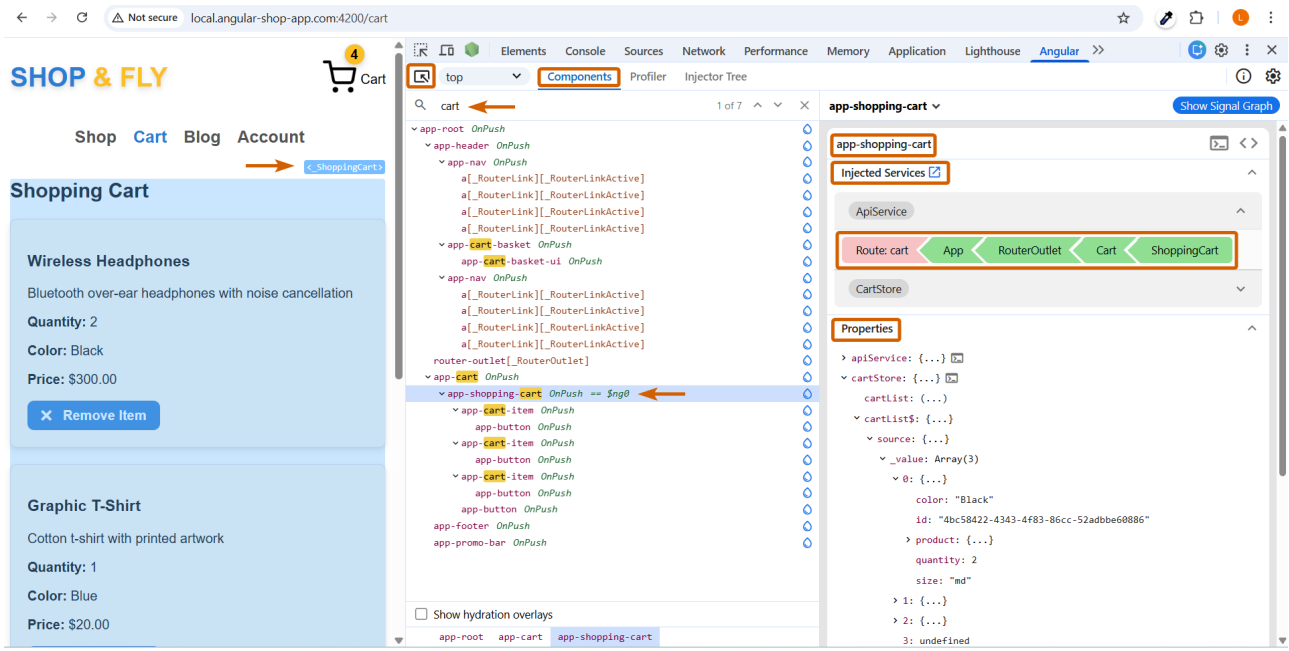
- **Chrome**: Chrome Web Store
- **Firefox**: Firefox Add-ons

Once installed, you'll see a new panel in your browser's DevTools named **Angular**. We will explore the following main tabs of the Angular DevTools extension: **Components**, **Profiler** and **Injector Tree**.

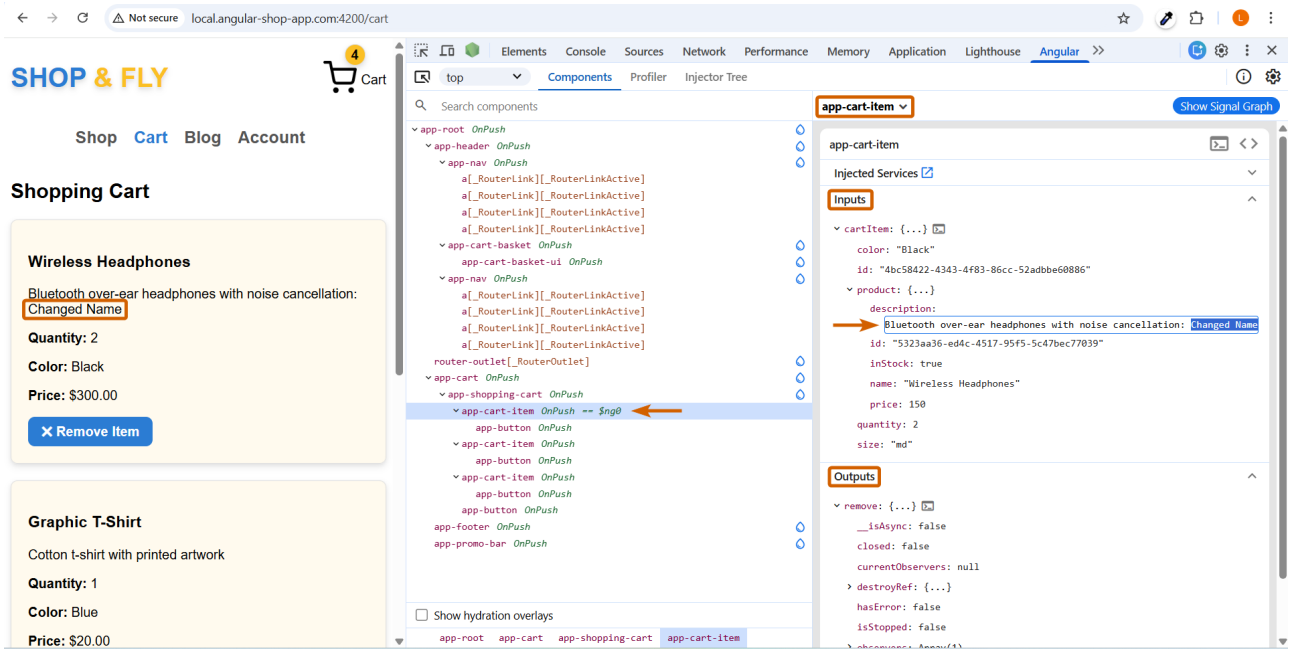
Angular DevTools: Components Tab

This tab provides an interactive view of your application's component tree, similar to how the **Elements** tab shows the DOM tree.

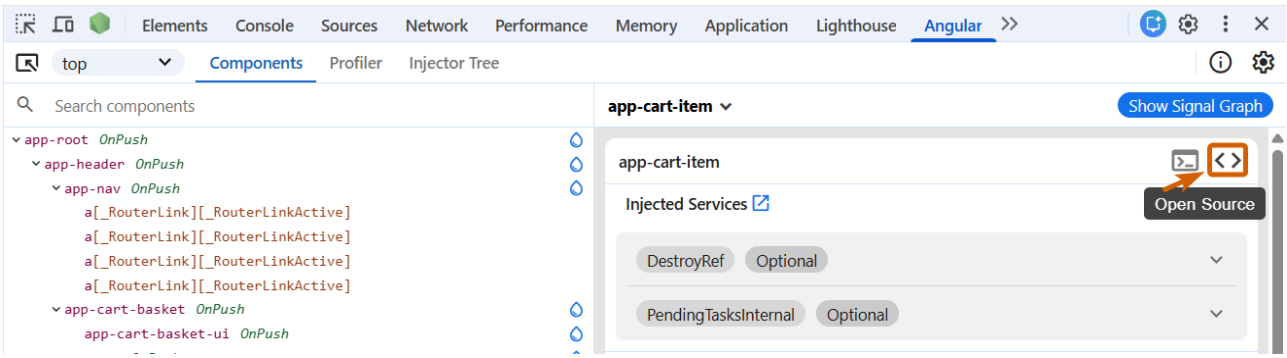
- **Inspect Component Tree**: Explore the hierarchical structure of your components as they are rendered on the page.
    - Click a component in the tree or the element selector (↖) icon to inspect it in the page.
    - Search components by name to make exploring faster and easier.
    - Click **Injected Services** to see the resolution path taken to resolve the dependency for the current component.
      This is particularly useful in large applications where identifying the exact source of a dependency is difficult. Without this feature, you would have to manually trace the provider hierarchy across multiple files in your source code, which makes debugging significantly harder.



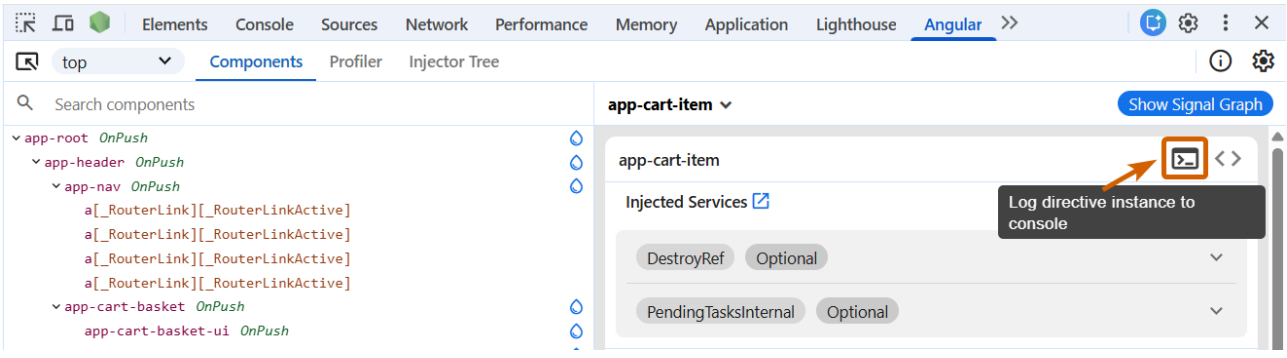
- **View and Edit State**: When you select a component, you can inspect its properties, inputs and outputs in the **Component Details** pane. Click a property value to enable the input and **modify it live** to instantly see how your component reacts.



- **Go to Source**: Click the **Open Source** (< >) icon in the **Component Details** pane to jump directly to its source code in the browser's **Sources** panel.



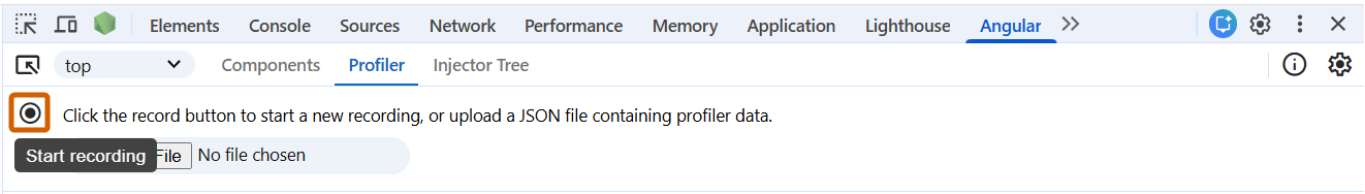
- **Log to Console**: Click the **Log directive instance to console** (>_) icon in the **Component Details** pane to log the selected component instance in the Console.



Angular DevTools: Profiler Tab

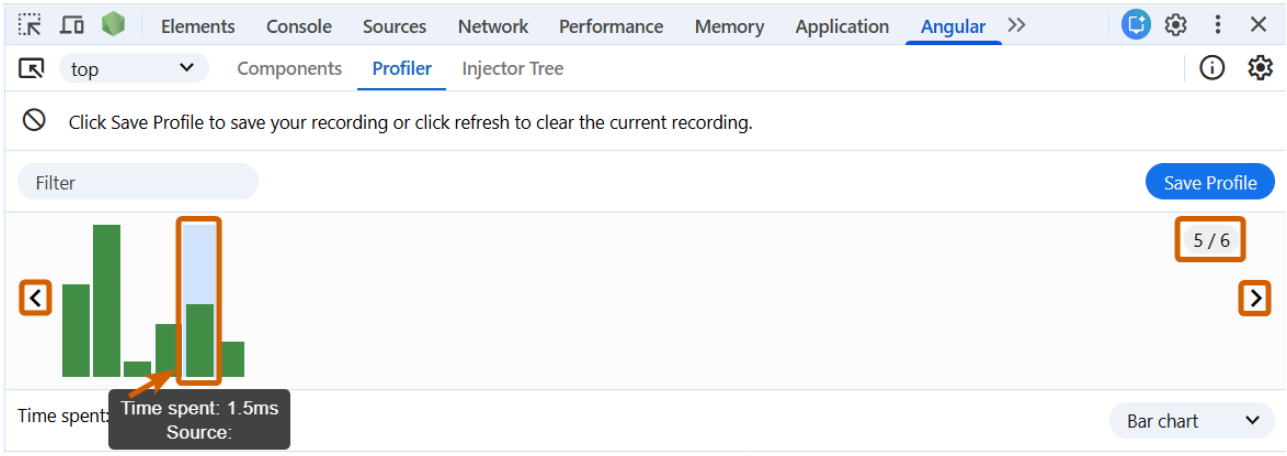
The Profiler is the primary tool for analyzing and optimizing your application's performance by focusing on **change detection**.

To record a profile, click the record icon (⦿) to start the recording, interact with your application and then click the stop (■) icon to stop the recording.

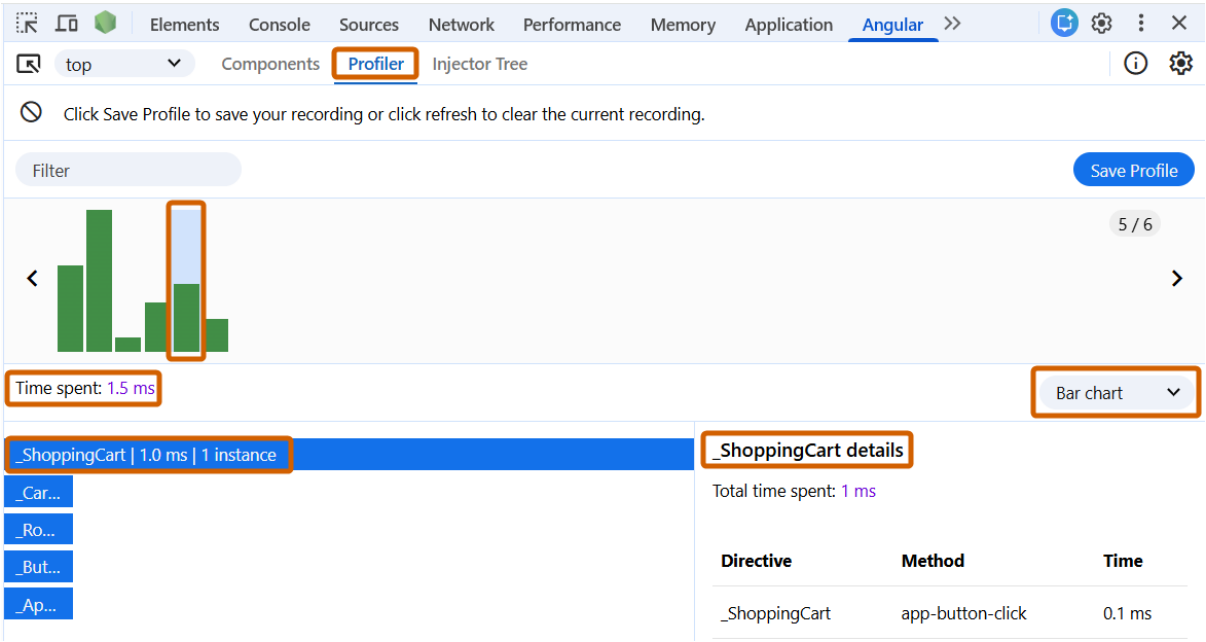


After recording is finished, you will have the following powerful features to debug your Angular application:

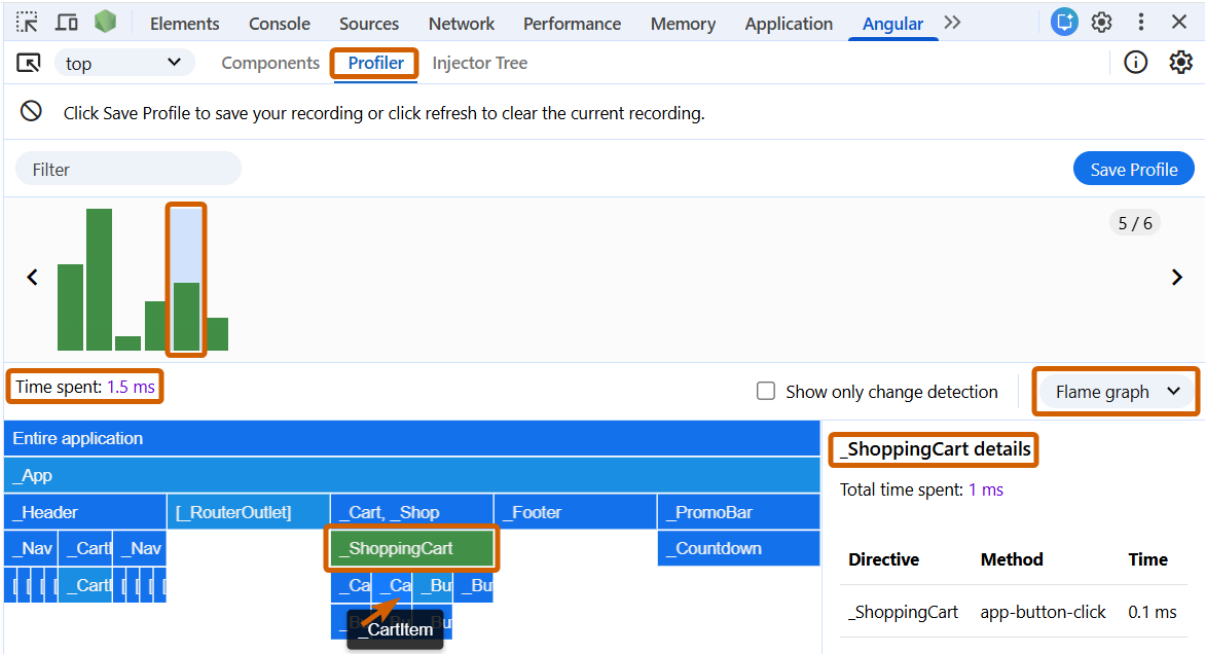
- **Visualize Change Detection**: The Profiler records all change detection events. Each bar in the timeline represents a **separate change detection cycle**, showing you how much time it took. You can use the Profiler timeline to understand if the current cycle caused any dropped frames (`>16ms`).



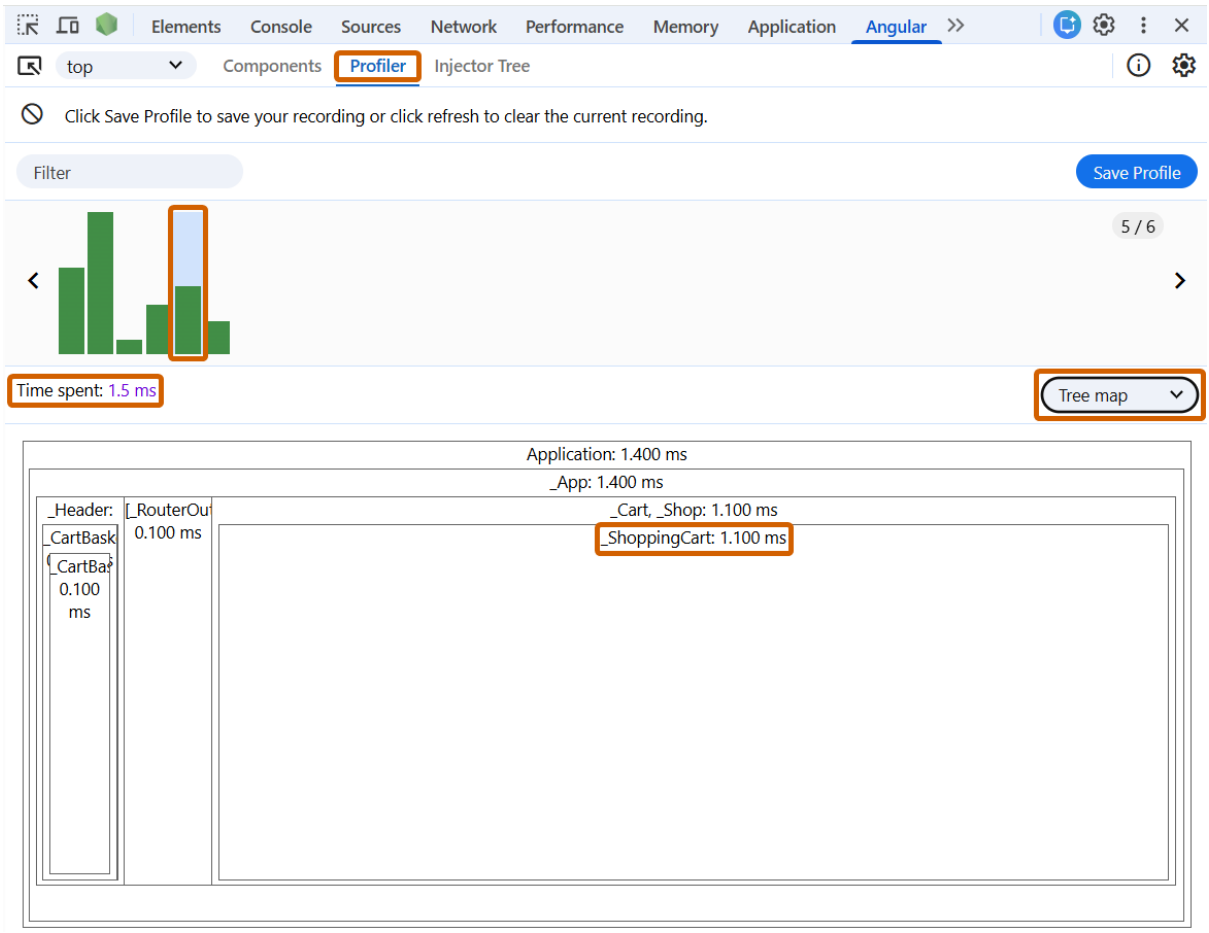
- **Alternative Views**: When you click a change detection bar, you can switch between three visualizations to better understand performance:
    - **Bar Chart**: The default view, showing each change detection cycle over time. Components in this view are sorted by execution time, with the **most expensive ones at the top**. You can click each high-cost component at the top and view additional details on the right panel.



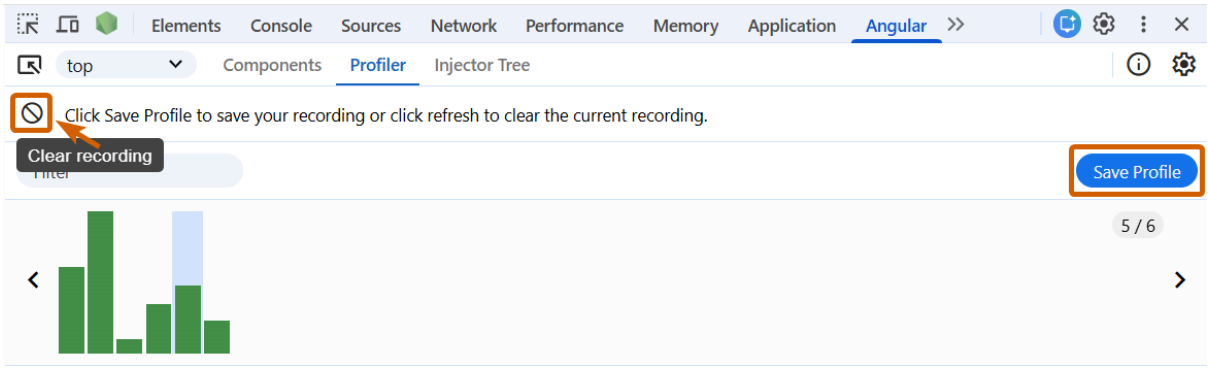
- **Flame Graph**: Displays components for the specific change detection cycle in the **context of the application tree hierarchy**, allowing you to **view parent-child relationships**. Components that were rendered during the current change detection cycle are highlighted in **different colors** (shades of green) and show a rendering timestamp when clicked. In contrast, other components that did not contribute to the change detection cycle are shown in blue and have `0 ms` as rendering time when clicking them.



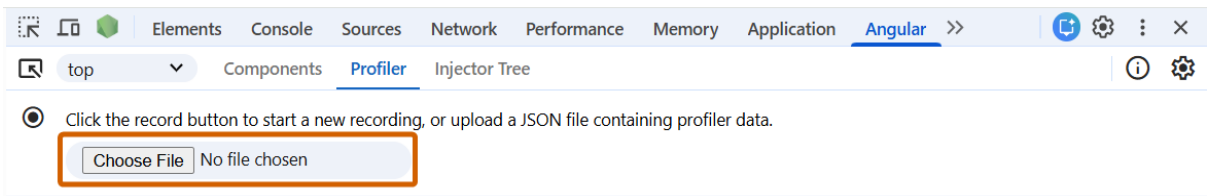
- **Treemap**: Displays components for the specific change detection cycle in a nested layout structure representing the application tree hierarchy. Shows at a glance how much each component rendering took and how the sum of child component rendering times contributes to the parent component rendering.
  For example, using this view and navigating through different change detection cycles, you can notice which child component contributes most to the rendering time of a specific parent component.



- **Save Profile / Load Profile:** Angular DevTools allows you to save the recorded profile in a **.json** file and load it later for further debugging.
    - After recording the profile, click the **Save Profile** button to download the profile as a **.json** file.



    - If you want to load a profile from a `.json` file, click the **Clear recording** icon to reset the Profiler. Once it is reset, you will see the **Choose File** button. Click it to attach the saved `.json` file to load the profile.



> **Note: Angular Change Detection** is the process through which Angular checks to see whether your application state has changed, and if any DOM needs to be updated. At a high level, Angular walks your components from top to bottom, looking for changes. Angular runs its change detection mechanism periodically so that changes to the data model are reflected in an application's view.

Angular DevTools: Injector Tree Tab

Angular Dependency Injection

Before exploring the Injection Tree tab in Angular DevTools, let's understand what dependency injection is in Angular.
Dependency injection (DI) is one of the fundamental concepts in Angular. DI is wired into the Angular framework and allows classes with Angular decorators, such as **Components**, **Directives**, **Pipes** and **Injectables**, to configure dependencies that they need.

- Angular facilitates the interaction between **dependency consumers** and **dependency providers** using an abstraction called **Injector**.
- When a dependency is requested, the injector checks its registry to see if there is an instance already available there. If not, a new instance is created and stored in the registry.
- Two main roles exist in the DI system: **dependency consumer** and **dependency provider**.
  In the example below, the `ShoppingCart` component is the dependency consumer and `ApiService` service is the dependency provider.

```
@Component({
    selector: 'app-shopping-cart',
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class ShoppingCart {
    private apiService: ApiService = inject(ApiService);
}
```

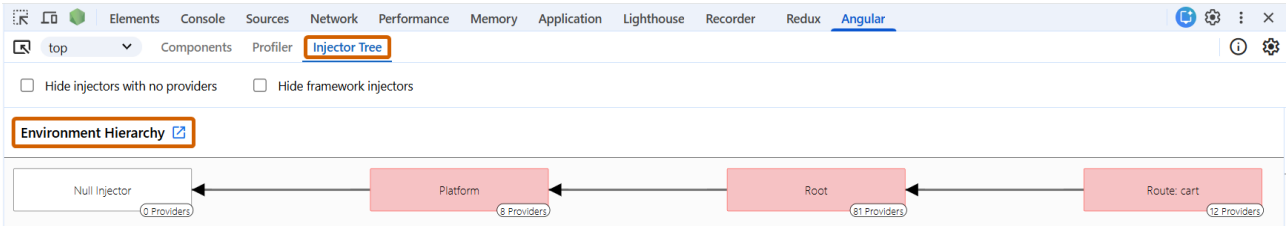
Angular has two injector hierarchies:

- **EnvironmentInjector hierarchy:** Configured using `@Injectable()` or `providers` array in `ApplicationConfig`.
- **ElementInjector hierarchy:** Created implicitly at each DOM element that matches a `Component` or `Directive`. An `ElementInjector` is empty by default unless you configure it in the `providers` property on `@Directive()` or `@Component()`.

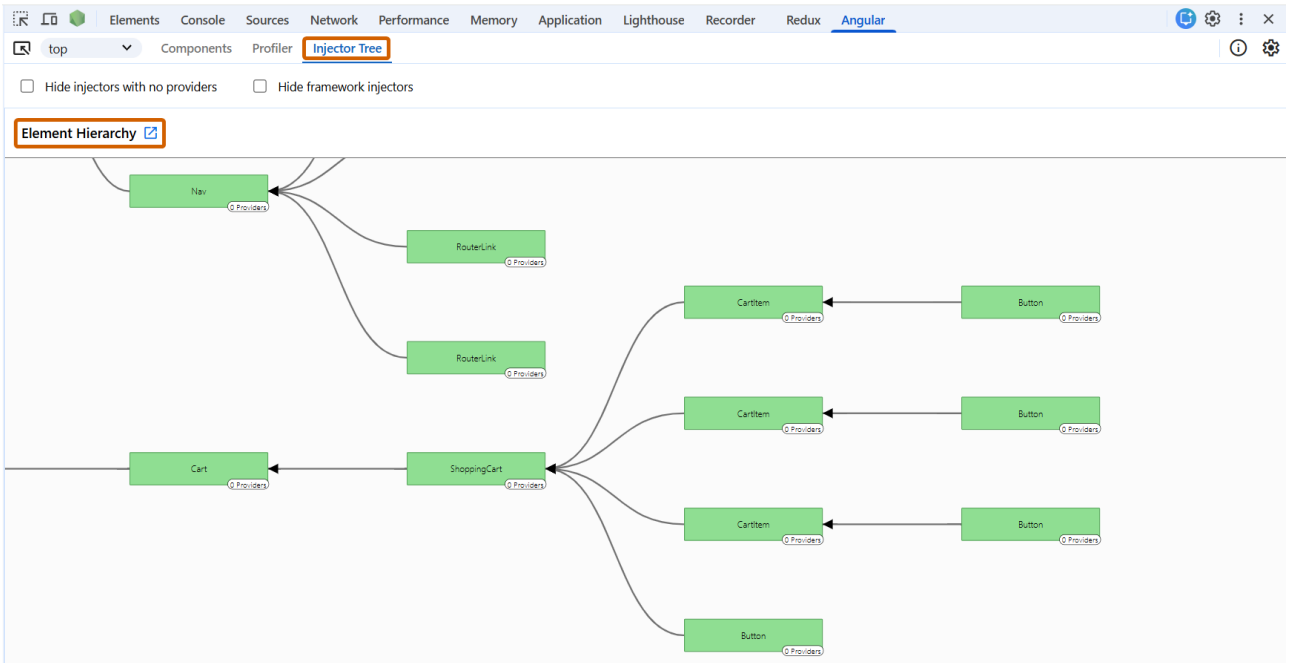
Injector Tree: Hierarchy Views

The **Injector Tree** tab helps you inspect the **Angular dependency injection system**. It shows how services and providers are resolved, making it easier to debug where a service is coming from. It provides two views:

- **Environment Hierarchy**: Shows the injector levels derived from the loaded environments in your Angular application.
    - An environment is created when your root component initializes. Examples of Angular environment injectors are `platform` and `root`, which will always exist in your view as they are created during bootstrapping.
    - Child environment injectors can be created when you dynamically load a chunk of your application (for example, via lazy-loading a component per route).



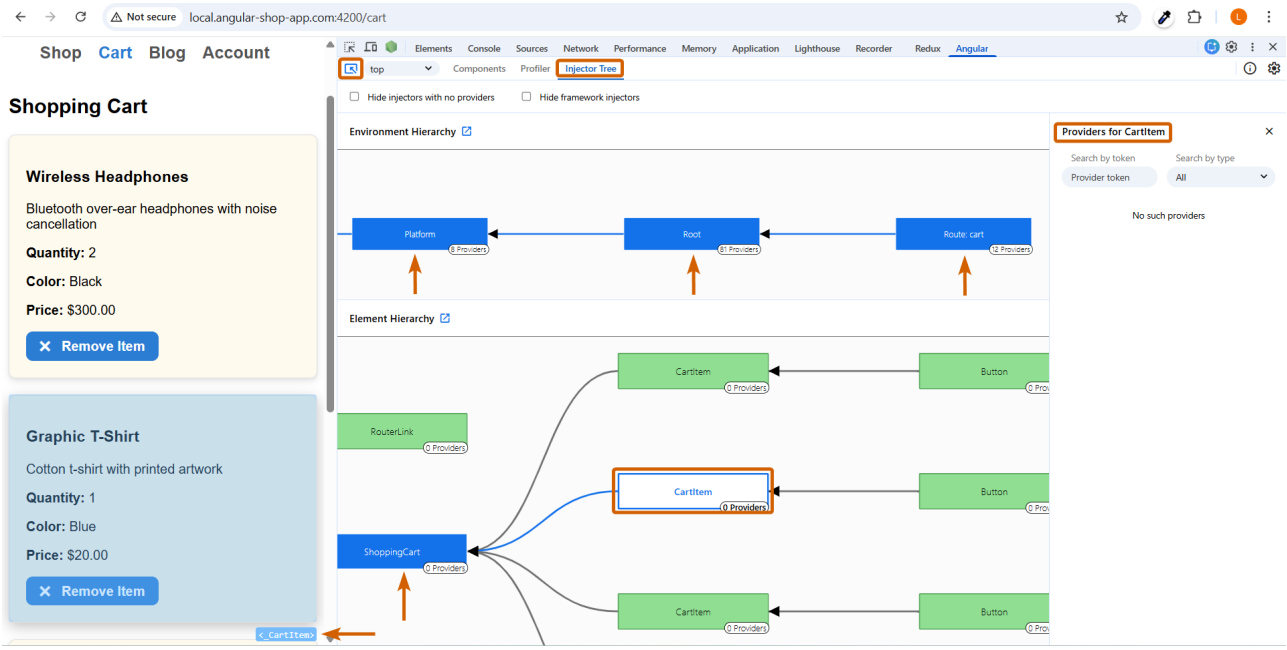
- **Element Hierarchy**: Shows the Angular component tree alongside the providers for each component to help you understand which components own which injectors. This is especially useful in large, complex applications when you need to understand the application dependency hierarchy at a glance and visualize the dependency relationships between components in different layers.



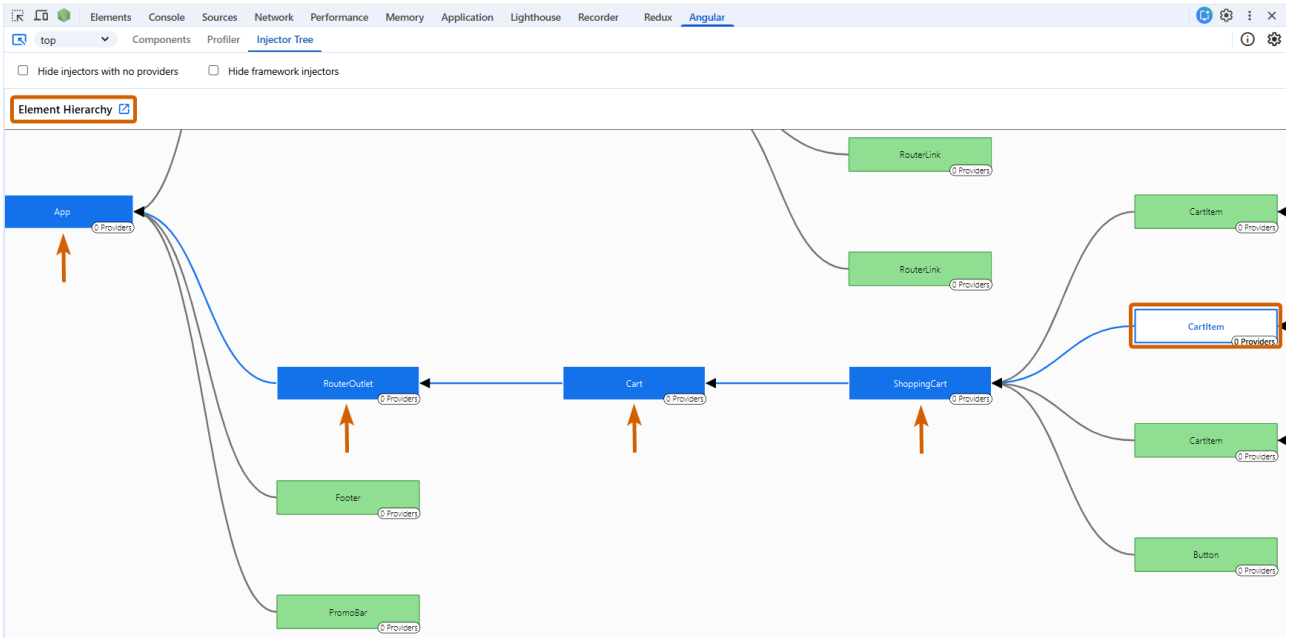
When resolving dependency injection for a component, Angular first traverses the **Element Hierarchy**, starting from the requesting component and **moving up towards the root component**, checking the `providers` array at each step. If it cannot find the provider in the component tree, it reaches the root of the component hierarchy and from there switches to checking the **Environment Injector** tree.

This is why you technically see the full Angular component hierarchy in the Element Hierarchy panel. If a component does not use specific providers, it will show as `0`. This simply indicates the provider is not found at this specific level, and the resolution path continues upwards (often resolving at the `root` environment level).

Since the element hierarchy can become very big and hard to navigate in large applications, you can use the inspector (↖) icon at the top-left of the Angular DevTools panel, to inspect an element in the page while keeping Injector Tree tab opened.
In this case inspecting an element in the page will reveal the corresponding component in the **Injector Tree Element Hierarchy view** and will also **highlight the exact element path** (in this example in blue color) allowing you to find the specific element and view its injectors' path. This is the **path that Angular takes to resolve the component's injectors**. That's why **Environment Hierarchy** is also highlighted in the same path color (`blue` in this case).



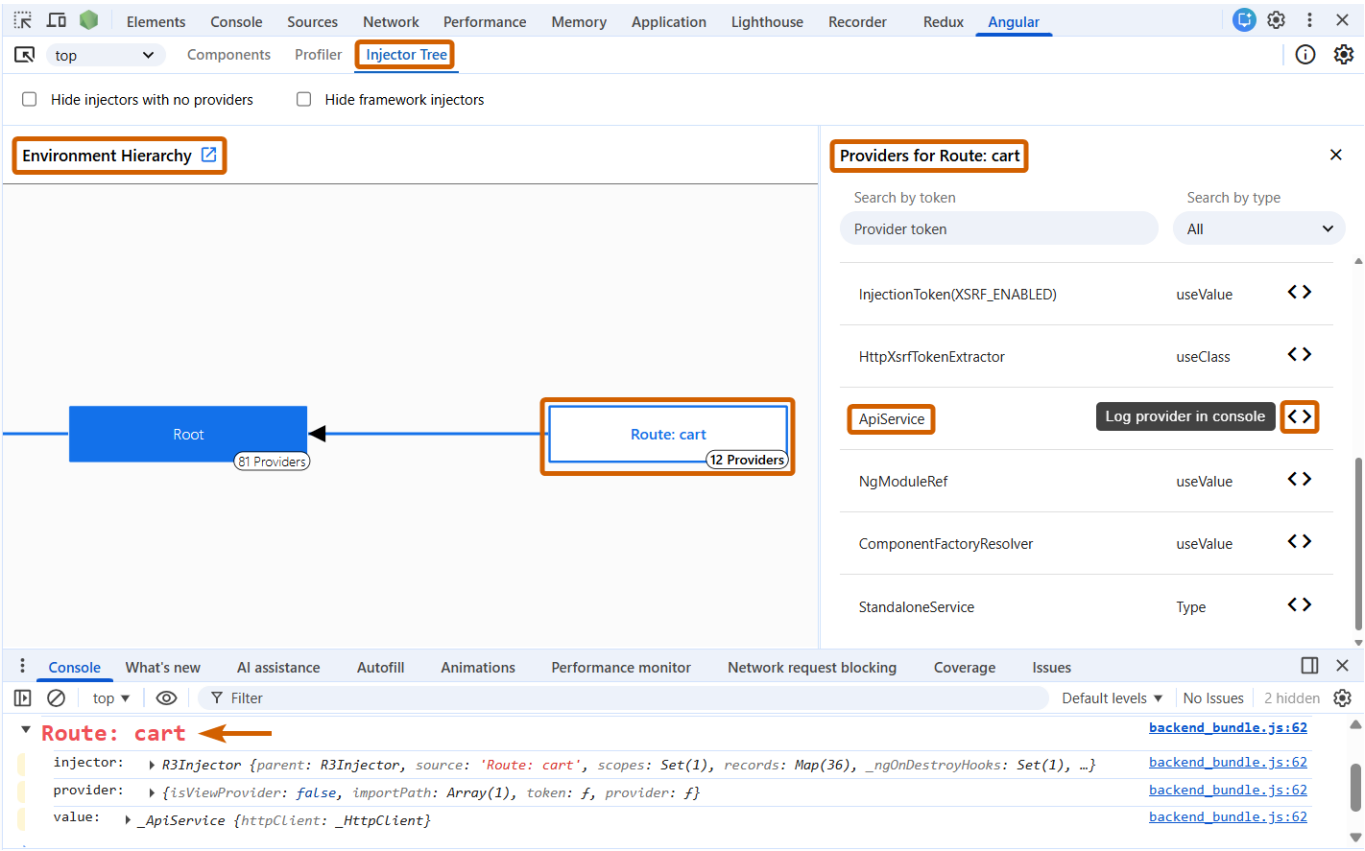
The full path highlight in Elements hierarchy view looks like this:



> **Note:** If you register your service with the root injector (via `providedIn: 'root'` or in the main application module's `providers` array), it remains a singleton for the entire application. Using `providers` at lower levels of the injector hierarchy (like inside specific components) will create additional local instances, altering the intended singleton scope.
> Technically, you only need to add a `providers` array to a lower-level component if you want to **break the singleton pattern** and create a new instance of a service specifically for that component (and its children). In that case, you will see the provider count reflect that in the tree.

Injector Tree: Inspect Details

Clicking each item in the Injector Tree tab allows you to view its providers on the right pane. Here you can review injection tokens and also console log resolved injection token results using **Log provider in console** (`< >`) icon.

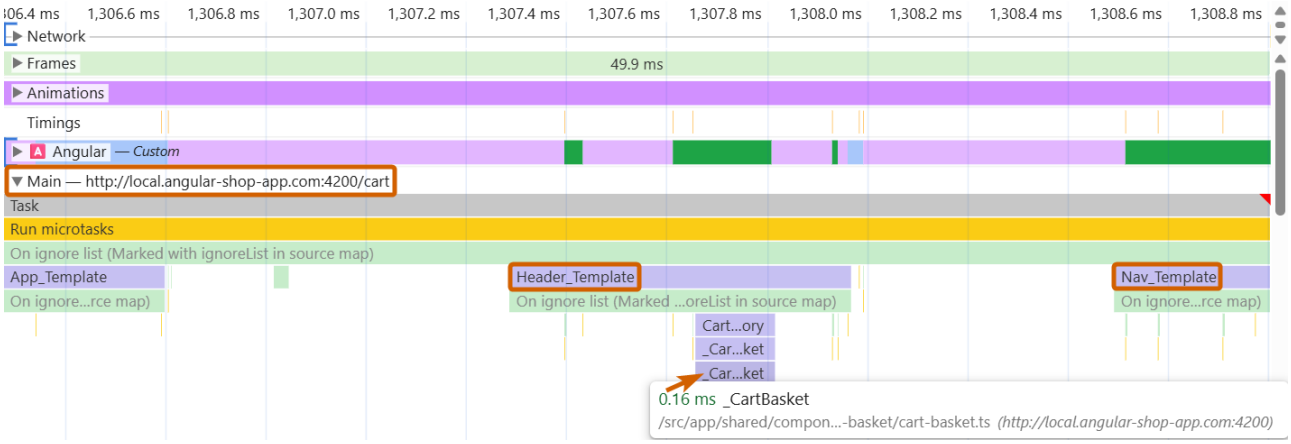


Angular Custom Performance Track

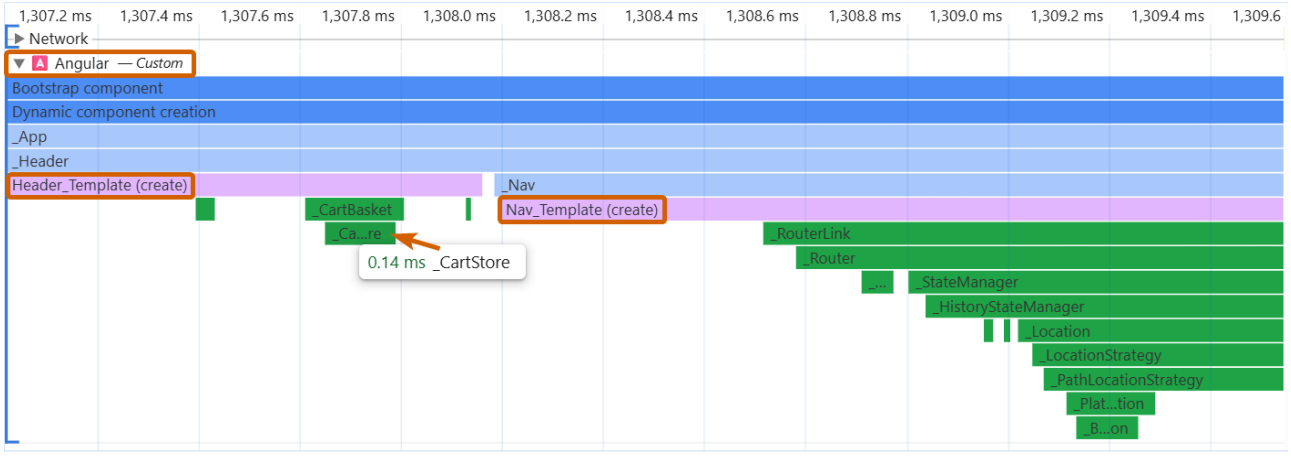
Angular integrates with the Chrome DevTools **Extensibility API** to present framework-specific data and insights directly in the **Chrome DevTools Performance panel**.

With the integration enabled, you can record a performance profile containing two sets of data:

- Standard performance entries based on Chrome's understanding of your code executing in the browser (Main thread flame chart).



- Angular-specific entries contributed by the framework's runtime (Angular Custom track).



Angular-specific data is expressed in terms of framework concepts (components, change detection, lifecycle hooks, etc.) alongside lower-level function and method calls captured by the browser.

By looking at different tracks in parallel, such as Angular data in the main thread (flame chart), the Angular Custom track and the Network track, you can see how Angular's application code is executed in the context of other browser processing. This allows you to review Angular data alongside typical layout and paint, other scripts, or specific network requests, which helps you understand and debug problems efficiently. This establishes a **unified debugging workflow**, reflecting a recent industry shift where frameworks provide a centralized experience that allows developers to review application performance from a single location: the browser **DevTools Performance panel**.

How to Enable Angular Custom Performance Track

You can enable Angular Custom Performance track in the DevTools Performance panel in one of two ways:

- Run `ng.enableProfiling()` in Chrome's Console panel or
- Include a call to `enableProfiling()` in your application startup code before `bootstrapApplication`.

Here is an example of how you can enable profiling in the application bootstrap to capture all possible events:

```
// main.ts
import { enableProfiling } from '@angular/core';
import { bootstrapApplication } from '@angular/platform-browser';
import { MyApp } from './my-app';

// Turn on profiling before bootstrapping your application
// to capture all the code run on start-up
enableProfiling();
bootstrapApplication(MyApp);
```

After integrating in one of the two ways mentioned above and recording a profile, you will see the **Angular Custom track**, with Angular tasks appearing within the existing tracks in the DevTools **Performance** panel.

> **Note:**
>
> - **Angular Custom Performance track** appears in Chromium-based browsers (e.g., Google Chrome, Microsoft Edge) in the DevTools **Performance** panel and is **separate from the Angular DevTools** extension.
> - Angular profiling works exclusively in development mode.

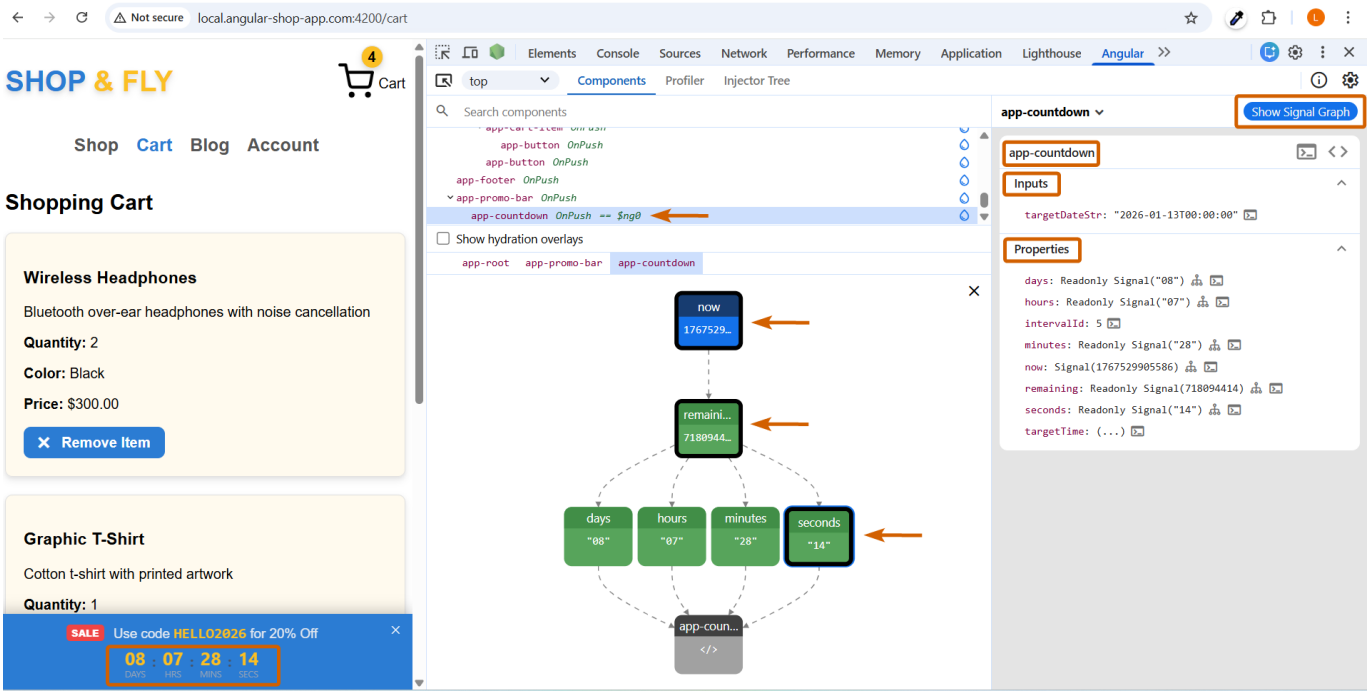
Angular DevTools: Debugging Signals

Angular DevTools provides a powerful way of debugging different types of signals, relationships between them and signal changes inside your component.

To debug signals inside the component, click on the specific component in the **Angular DevTools > Components tab**. Click the **Show Signal Graph** button on the right pane. You will see a new dialog appear right below the **Element Hierarchy** pane where you can view all signals, their live updates and relationships.

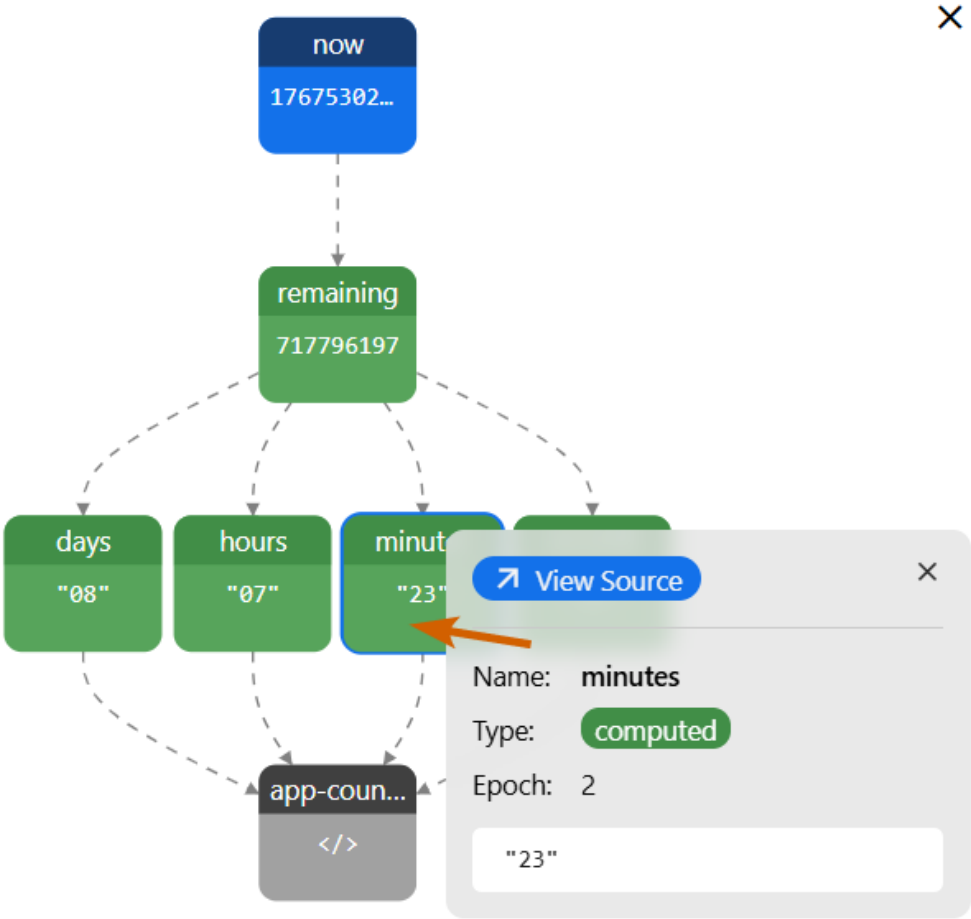
For example, let's imagine you have a promo-code banner which shows a live time countdown until the promo code expires. As you can easily see from the graph:

- `now` is a `WritableSignal` (color **blue**), which shows the current timestamp.
- `remaining` is a `computed` signal (color **green**) which is calculated based on `now` and shows the remaining timestamp.
- `days`, `hours`, `minutes` and `seconds` are `computed` signals which are calculated based on the `remaining` signal.
- And finally all the signals are listed in the `app-countdown` component template.



Here you can have these valuable actions:

- Click on each signal to view more information about it:
  - **`Type`**: Shows the type of the signal such as `computed` or `signal` (`WritableSignal`).
  - **`Epoch`**: Shows how many times the signal has changed since the app started.
  - **`Current value`**: Shows the current value of the signal.



- When the signal value is changed, you will see it **blinking** in the signal graph. During **blinking**, the signal box is highlighted with a thick border which helps you easily notice updates.
  In our case, `now` and `seconds` signal values are changing once per second, so they are blinking once per second.
  The **highlighting signal updates** feature is very useful, for example, it helps you notice computed signals that are not updated appropriately based on dependent signals and spot potential bugs.

Angular DevTools: Debugging Hydration

What is Hydration

Before jumping into hydration debugging, let's understand what Angular's rendering strategies are and what hydration is.

Rendering strategies determine when and where your Angular application's HTML content is generated. Angular supports three primary rendering strategies:

- **Client-Side Rendering (CSR):** The page is rendered entirely in the browser.
- **Static Site Generation (SSG/Prerendering):** The pages are pre-built at build time on the server and served as static files.
- **Server-Side Rendering (SSR):** The server generates the HTML for the page and sends it to the browser to render upon receiving a page request from the browser.

Rendering content on the server side gives a huge boost in performance and decreases page initial load times (improving `LCP`). Unlike client-side rendering, these methods generally provide excellent SEO, as search crawlers receive a fully rendered HTML document. However, they also have limitations, such as the requirement to author code that does not strictly depend on browser APIs, which limits your selection of JavaScript libraries to those that expect to run in a browser.

For the page received from the server to become interactive, there is a need to attach JavaScript behaviors to a server-rendered or statically-generated HTML page. This process is called **hydration**. Once the browser displays the server-rendered HTML, Angular hydrates the application. It then runs entirely in the browser like a traditional SPA: subsequent navigation, route changes and API calls all happen client-side without additional server rendering.

Let's understand **Server-Side Rendering (SSR)** and hydration better with the **Angular Shop** POC example. It has `Shop` (default route), `Cart`, `Blog` and `Account` routes, and all of them are SSR enabled as specified in the `app.routes.server.ts` file:

```
export const serverRoutes: ServerRoute[] = [
    {
        path: '**',
        renderMode: RenderMode.Server
    }
];
```

`provideClientHydration` is enabled in `app.config` for routes and for the application itself:

```
export const appConfig: ApplicationConfig = {
    providers: [
        provideClientHydration(
            withHttpTransferCacheOptions({
                includePostRequests: true
            }),
            withEventReplay()
        ),
        provideBrowserGlobalErrorListeners(),
        provideZonelessChangeDetection(),
        provideRouter(routes)
    ]
};
```

Before debugging, open the **Network** panel so that it is easy to understand the flow. For this setup, the following behavior is observed:

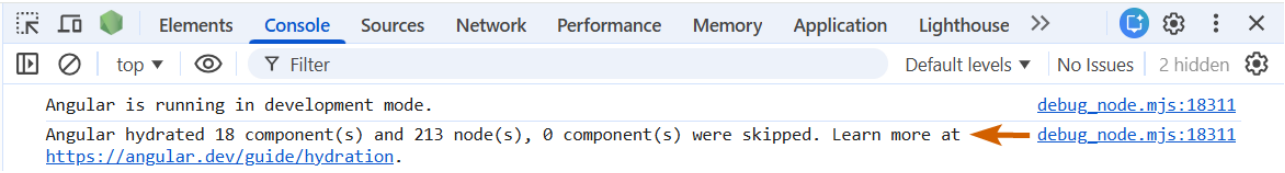
- If you refresh the application on a specific route (e.g., `Cart` page), the content will be rendered on the server side and then hydrated. If you check your Network panel, you will not see any API calls on the client side, and you will also not see any loading skeleton on the UI. This is because the rendering happened on the server side.
- Let's explore another scenario. If you navigate first to the `Shop` page and refresh it, it will be rendered on the server side. From here on, the application is hydrated, and it behaves as a single-page application. This means if you go to any SSR-enabled route, for example `Cart`, you will see a loading skeleton and API request in your Network panel.
- You can easily debug this by adding a `console.log` or debugger breakpoint in your IDE on the SSR server file: `server.ts` inside `app.use()`.

The idea is that SSR rendering happens only for the current specific route, and subsequent navigations already happen on the client side. That's why it is more useful for **SEO** and **LCP** optimizations, delivering the first critical content to users as soon as possible.

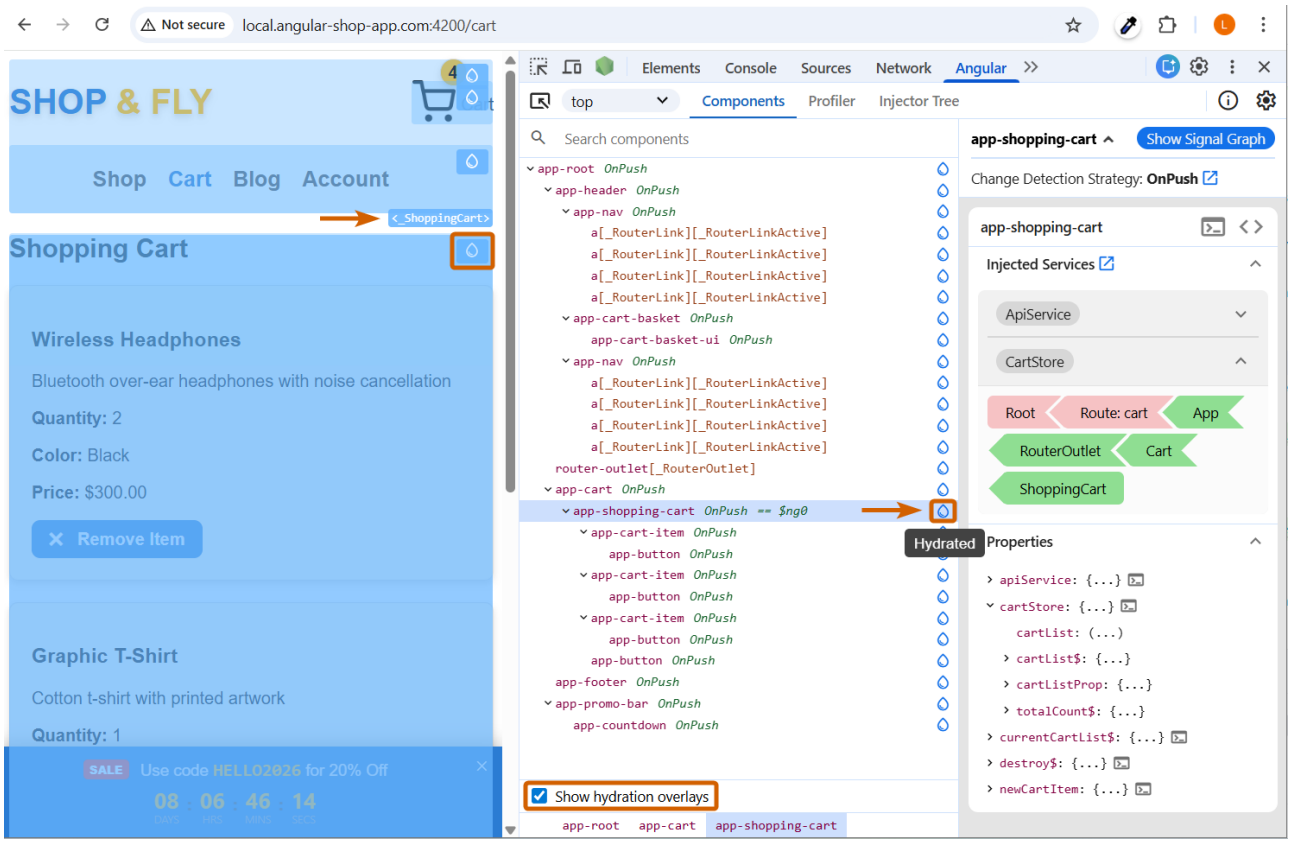
How to Debug Hydration

You can debug hydration using the Console panel and Angular DevTools:

- **Debugging Hydration in the Console panel:** If after an application refresh you check the Console, you will see a message that includes hydration-related stats, such as the number of components and nodes hydrated. Angular calculates the stats based on all components rendered on a page, including those that come from third-party libraries.



- **Debugging Hydration with Angular DevTools:** Angular DevTools allows you to effectively debug hydration:
    - If the page is hydrated, you will see a little blue hydration (💧) icon next to each hydrated component in the **Component Tree** pane.
    - You can also check the **Show hydration overlays** checkbox at the bottom of the **Component Tree** pane, which will highlight all hydrated components with overlays so you can easily spot them. This will also help you spot any hydration issues: if an element is not highlighted in the overlay, you can understand that hydration failed.



Angular DevTools: Debugging Incremental Hydration

What is Incremental Hydration

Incremental hydration is an advanced type of **hydration** that can keep sections of your application dehydrated and **incrementally trigger hydration on them only when needed**. It is a performance improvement that allows you to produce smaller initial bundles while still providing a full application hydration experience. Smaller bundles improve initial load times, reducing **First Input Delay (FID)** and **Cumulative Layout Shift (CLS)**.

You can enable incremental hydration for applications that already use server-side rendering (SSR) with hydration by adding the `withIncrementalHydration()` function to the `provideClientHydration` provider.

```
export const appConfig: ApplicationConfig = {
    providers: [
        provideClientHydration(
            withHttpTransferCacheOptions({
                includePostRequests: true
            }),
            withIncrementalHydration()
        ),
        provideBrowserGlobalErrorListeners(),
        provideZonelessChangeDetection(),
        provideRouter(routes)
    ]
};
```

Incremental hydration builds on top of full-application **hydration**, **deferrable views** and **event replay**. With incremental hydration, you can add additional triggers to `@defer` blocks that define incremental hydration boundaries. You can specify **hydration triggers** that control the conditions for when Angular loads and hydrates deferred content. For example, the **`hydrate on`** trigger can help you to specify a `@defer` block to be hydrated on different events: **`hydrate on viewport`**, **`hydrate on hover`**, **`hydrate on timer`**, etc.

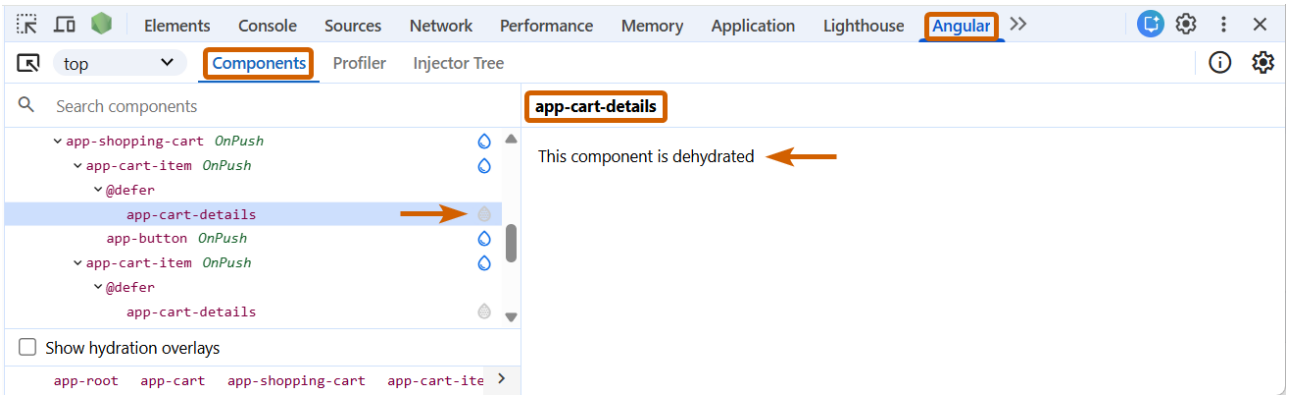
In the example below, the `app-cart-details` component is deferred within `app-cart-item` component when it appears in **viewport**.
Below is the code block in the `app-cart-item` component template:

```
@defer (hydrate on viewport) {
  <app-cart-details [details]="cartItem.product.details?? null" />
} @placeholder {
  <div>Loading Details...</div>
}
```

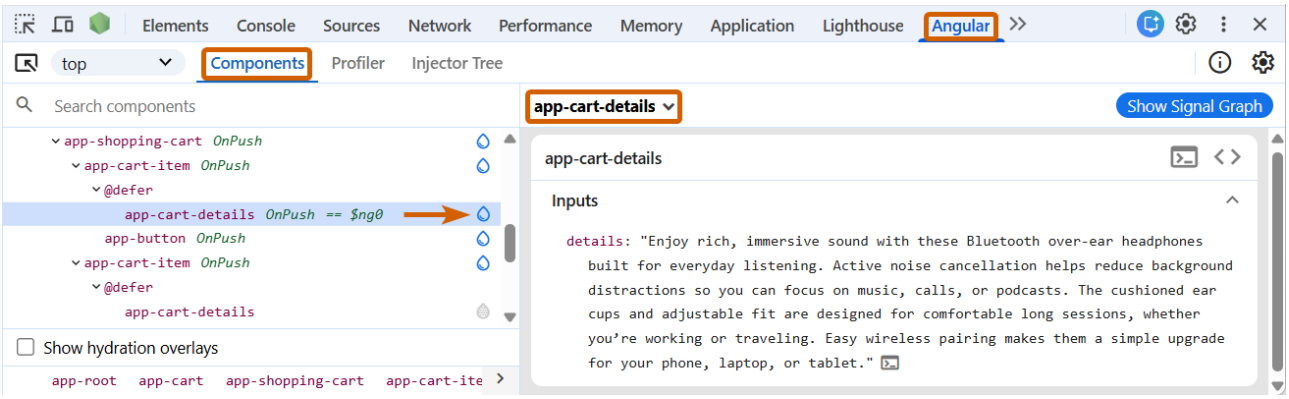
How to Debug Incremental Hydration

Let's use Angular DevTools to debug the example discussed above where the `app-cart-details` block is deferred within the `app-cart-item` component when it appears in the viewport.

- First run the development server with the `ng serve --no-hmr` option.
- Open the **Angular DevTools > Components** tab and find the element that has not yet entered the viewport. You can see a small grey hydration icon (💧) next to the component within the `@defer` block, which indicates the component is dehydrated.



- Now scroll to that specific component to trigger hydration. You will see that the active blue hydration icon (💧) appears next to the component within `@defer` block. Now the component is hydrated and available as an Angular component. You can click on it and view the details on the right pane, like for any other component.



Note:

- **IMPORTANT:** To test incremental hydration in local development build, make sure to run it with HMR disabled option: `ng serve --no-hmr`. This is because when hot module replacement is enabled, all defer blocks and their dependencies are loaded eagerly.
- Incremental Hydration depends on and enables **event replay** automatically. If you already have `withEventReplay()` in your list, you can safely remove it after enabling incremental hydration.

Angular DevTools: DevTools and Production Builds

Angular DevTools **only works on development builds**. Production builds strip out the required debug information to optimize for size and performance.

To use Angular DevTools, you must run your application in a development configuration:

- **Local Environment:** Running your application locally using `ng serve` will automatically enable Angular DevTools.

- **Deployed:** Use `ng build --optimization=false` to build production assets in a way that they are compatible with Angular DevTools. Remember to do this only in lower environments (e.g., **testing** and **staging**) and avoid it in the production environment.

A safe way to enable Angular DevTools capability in lower environments is to specify two different commands in `package.json`:

```
{
  "scripts": {
    "build:prod": "ng build --configuration=production",
    "build:debug": "ng build --configuration=production --optimization=false"
  }
}
```

Then in your **Dockerfile** used by your **CI/CD pipeline**, read the current environment variable (e.g., `prod`, `test`, `staging`) and call the corresponding command. For example, you can use `build:debug` for testing and staging environments and `build:prod` for the production environment to keep it optimized and safe.

Angular DevTools: Common Use Cases

- **Debug SSR Hydration:**
    - Use the **Components** tab to check which components were hydrated or incrementally hydrated.
    - Use the **Show hydration overlays** action to display overlays over hydrated components and spot which components skipped hydration.
- **Debug Signal Relationships:**
    - Use the **Show Signal Graph** action in the **Components** tab to view the signal tree for the specific component, signal relationships, signal updates as well as additional details about each signal.
    - The feature of **highlighting signal updates** helps you spot bugs when certain signals are not updated as part of a specific action.
- **Identify Dropped Frames and Slow Interactions:** By recording a profile and using the **Bar chart**, you can identify the most expensive components. If the time spent rendering a frame is more than **16ms**, it indicates a dropped frame and is a strong candidate for optimization.

  You can use the **Flame chart** to identify parent/child relationships and understand what caused the high rendering time. Below are common scenarios that cause dropped frames or **jank** experiences (visual stuttering):
    - **Redundant Calculations**: This occurs when a parent component (e.g., `Cart`) has nested components (e.g., `CartItem`) that rely on values that should only be recalculated when a

specific item changes (like `price`), but are instead recalculated upon any list update.
Debug Tip: In Angular DevTools, check which components rendered when adding or removing an element. If the entire list re-renders, optimizations may be needed.
Solution: Use `ChangeDetectionStrategy.OnPush`, Pure Pipes or memoization utility libraries (e.g., **Lodash**). In modern Angular, use **Signals**: a `computed` signal provides built-in memoization and only recalculates when its specific dependencies change.

- **Large Component Tree**: Even if individual components are optimized, rendering thousands of components in a single view can overwhelm the browser (e.g., listing a lot of products in the **Shop** page). For every update, Angular may run change detection on the whole list, causing dropped frames.
  Debug Tip: Check the total time the parent component took to render in the Profiler. If that time is high and is a computed time for thousands of child component render times, this is a good candidate for optimization.
  Solution: Reduce the DOM size using **Pagination**, **Load More** buttons or **Virtual Scrolling** (rendering only the items currently visible in the viewport).

Important Debugging Tip: Never over-optimize. For example, in some cases rerendering the whole list for one small action could be fine and not cause any performance issues. If you follow Angular engineering best practices, you can skip these kinds of scenarios and apply optimizations only if you notice real degradation in performance.

- **Inspect Component Hierarchy and Component Details:**
  - Use the **Components** tab to inspect the hierarchical structure of your components as they are rendered on the page and view each component's inputs, outputs and properties as you click the component.
  - Make live updates on components in the **Component Details** pane to experiment with different scenarios and reproduce bugs easily.
- **Debug Dependency Injection:** Use the Angular DevTools **Components** and **Injector Tree** tabs to debug dependency injection of your Angular application.
  - Use the **Injected Services** section in the **Components** tab to quickly view the dependencies of the currently selected component. Clicking on each dependency shows the resolution path that was taken to resolve the dependency for the current component.
  - Use the **Injector Tree** tab to debug the injector hierarchy for environment injectors (`platform`, `root`, lazy loaded modules) and your Angular components and directives. For example, this can help to quickly identify where a specific service is provided and notice any unnecessary instances that break the singleton pattern.

This is a very powerful feature because dependency injection in Angular has always been hard to debug. Without Angular DevTools, it would take a lot of work to manually check in your source code and find out dependencies for each component.

- **Export Profiler Trace and Attach to Bug Ticket:** Use the **Save Profile** button in the **Profiler** tab to save the debugging steps that led to a bug situation in `.json` format. Attach it to the bug ticket and send it to a team member for further review.

> **Note**: You can also explore the new **Router Tree** tab in Angular DevTools which recently became stable. It allows you to visualize the hierarchy of your application's router logic and lazy-loaded components.

4.4 Redux DevTools

Redux DevTools is a powerful developer tool for debugging applications that use a centralized state management pattern like **Redux** or any other architecture which handles the state change (through custom integrations).

It provides a detailed view of your application's state, the actions that change it and the ability to **time travel** through those changes. While it can also be used as a **standalone application** or as a **React component integrated into the client application**, in this section we will discuss **Redux DevTools browser extension**.

The **Redux DevTools browser extension** is available in major browsers:

- **Chrome**: Chrome Web Store
- **Firefox**: Firefox Add-ons
- **Edge**: Microsoft Edge Add-ons

Once installed, you'll see a new panel in your browser's DevTools named **Redux**.

In this section, we will explore Redux DevTools with a simple **React Todo application** example with a **Redux store** that has the following actions:

- **todoAdded:** Updates the list of todos after the `insert` operation.
- **todoDeleted:** Updates the list of todos after the `delete` operation.
- **loadingStarted:** Indicates when loading of todos started.
- **loadingCompleted:** Indicates when loading of todos completed successfully.
- **loadingFailed:** Indicates when loading of todos failed.

Redux DevTools: What is Redux

Before starting to debug our application with Redux DevTools, we need to understand the core concepts of state management and the Redux pattern.

State Management Terminology

- **State**: A piece of data that represents the current snapshot of your application and determines its behavior or appearance.
- **Store**: A centralized container that holds state, provides rules for updating it and allows components to access state or react to its changes.

Redux Library

Redux is a JavaScript library for predictable and maintainable global state management inspired by the Flux pattern. While initially adopted in the React ecosystem, the core Redux library itself is a standalone JavaScript library and can be used with any UI layer, not just React. Specialized **UI binding** libraries handle the integration logic (e.g., **react-redux**). Additionally, Redux's robust architecture has led to the integration of its pattern with many other libraries and frameworks, including **Angular** (with **NgRx**).

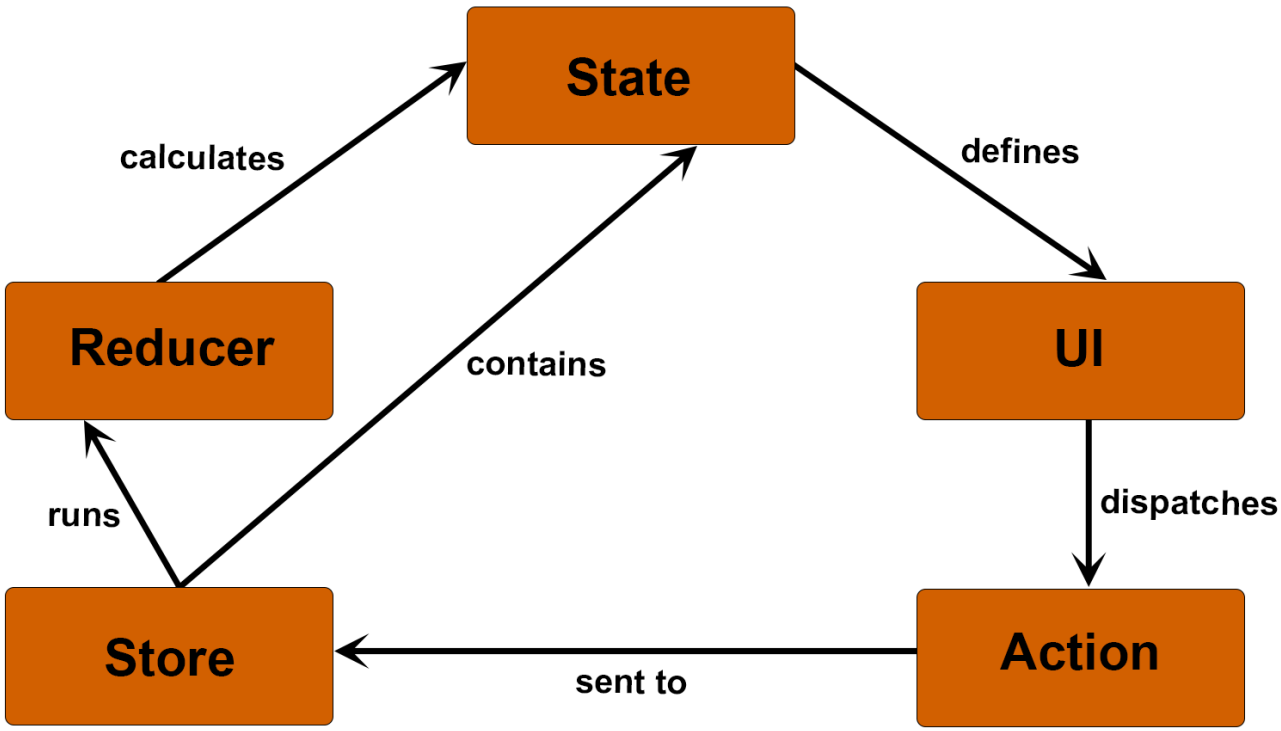
Redux can be described in three fundamental principles:

1. **Single source of truth:** The global **state** of your application is stored in an **object tree** within a **single store**.
2. **State is read-only:** The only way to change the state is to emit an **action**, an object describing what happened.
3. **Changes are made with pure functions:** To specify how the state tree is transformed by actions, you write **reducers** which are **pure functions** that take the previous state and an action and return the next state.

This is a small example of **one-way data flow** (e.g., **adding a new todo**) to understand how Redux works:

- **State** describes the condition of the application at a specific point in time and lives in the **store**.
- The **UI** is rendered based on that state.
- When something happens (e.g., **a new todo is added**), the UI dispatches an **action** (e.g., `todoAdded`) and provides the payload of the data to be updated.
- The action is sent to the **store**, and the **store** runs the **reducer** function to calculate a new state.
- The UI re-renders based on the new state.

Check the diagram below:

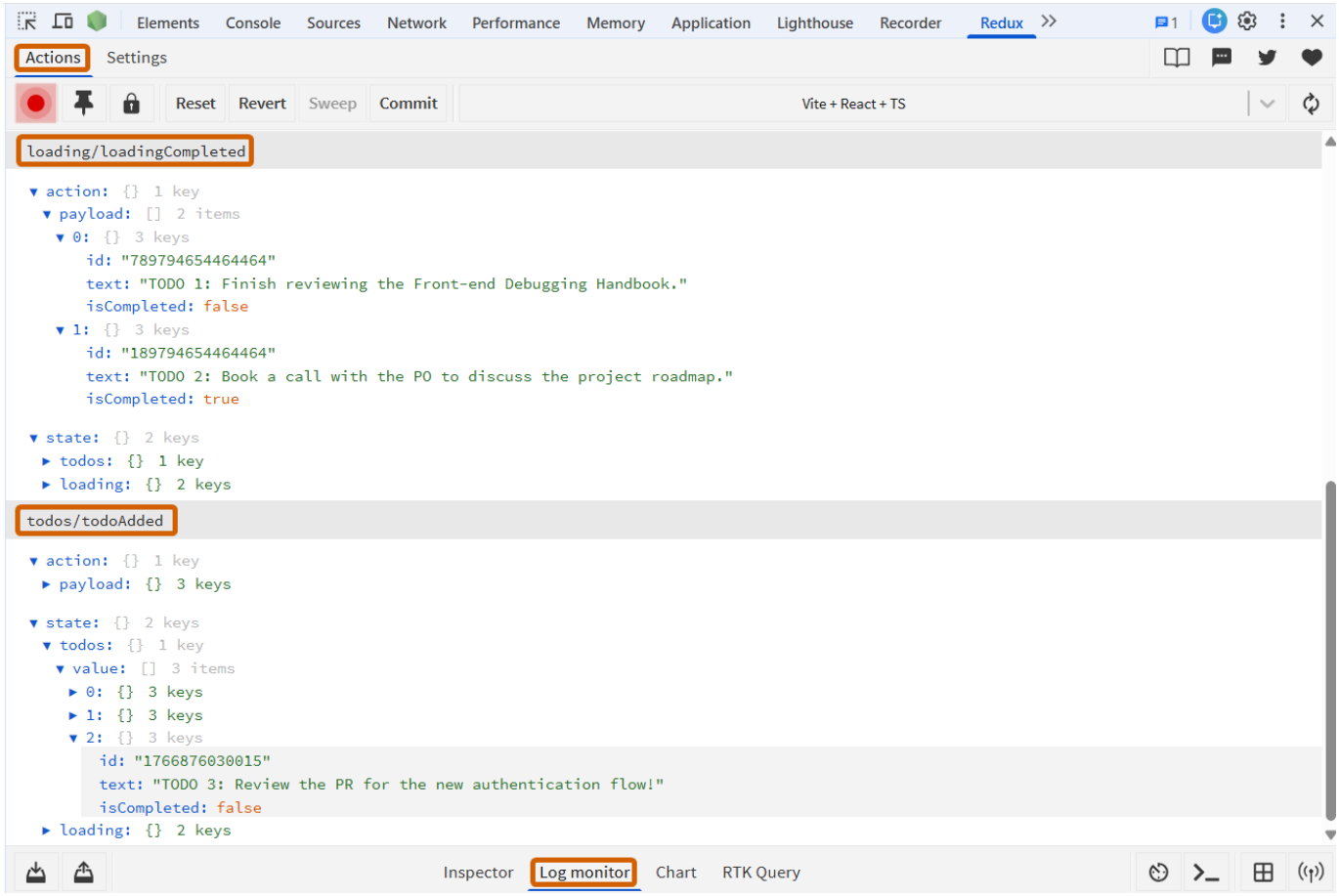


Redux DevTools: Core Monitors

To inspect your application state, Redux DevTools provides the following core monitor types, which you can switch using the toolbar at the bottom of the Redux DevTools panel:

- **Inspector:** The most powerful and frequently used monitor. It shows a list of actions (their types) on the left pane. For each action, it allows you to inspect the payload in different formats, view traces, visualize the difference from the previous version, and even generate a unit test based on the current state snapshot.
- **Log Monitor:** Shows a vertical list of actions and the resulting state tree. It is useful for a quick glance at the history but can become cluttered in large applications.
- **Chart:** Renders your state as a visual tree diagram. This is useful for understanding the shape of your store if it has many deeply nested slices.
- **RTK Query:** A specialized monitor for debugging server state that visualizes active queries, request statuses, mutations and cache management. The monitor shows data only if you are using Redux Toolkit's built-in data fetching.

Below is an example of what the **Log Monitor** looks like after adding a new todo:



Since **Inspector** is the most useful tool for debugging your application state, we will explore it in detail in the next section.

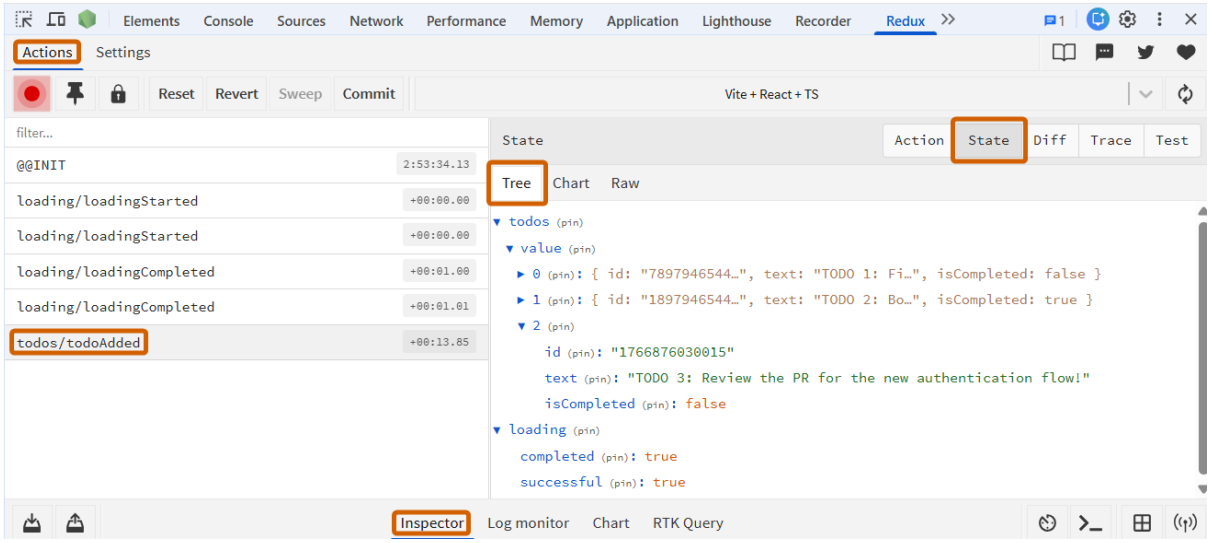
Redux DevTools: Inspector Monitor

The **Inspector Monitor** separates the screen into two parts:

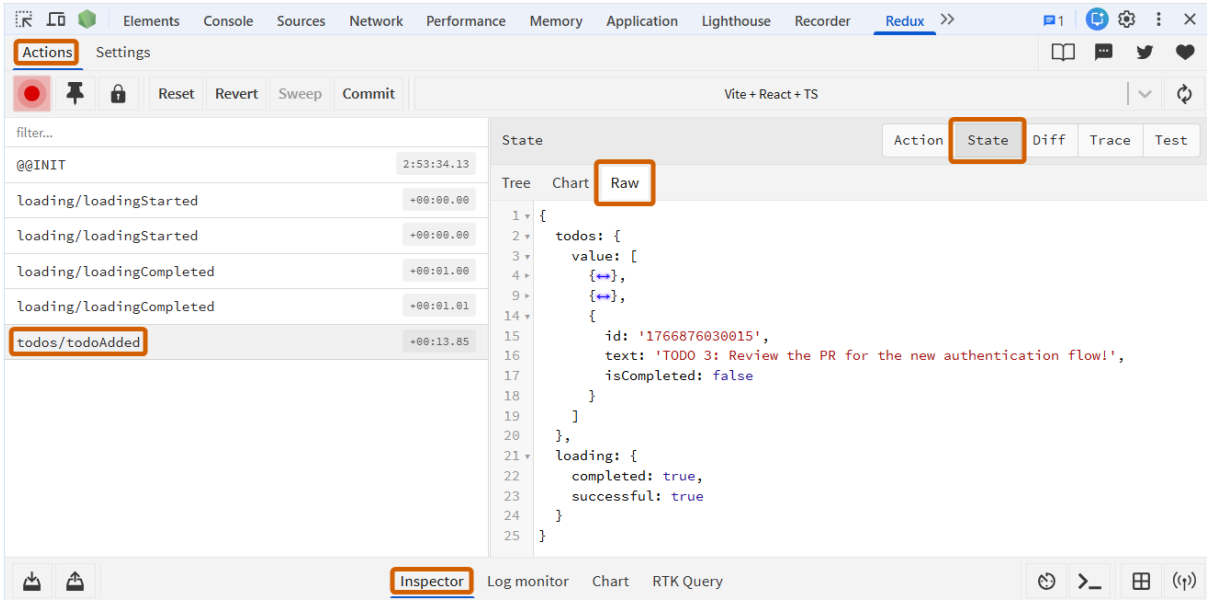
1. Action List: This panel on the left shows a real-time, chronological log of dispatched actions. It displays the action type as a unique identifier for the action (e.g., `todos/todoAdded`, `loading/loadingCompleted`).

2. State and Action Inspector: This panel on the right allows you to inspect the dispatched action details and the entire application state tree at the specific point in time when the action was dispatched. Below are the useful features of this panel:

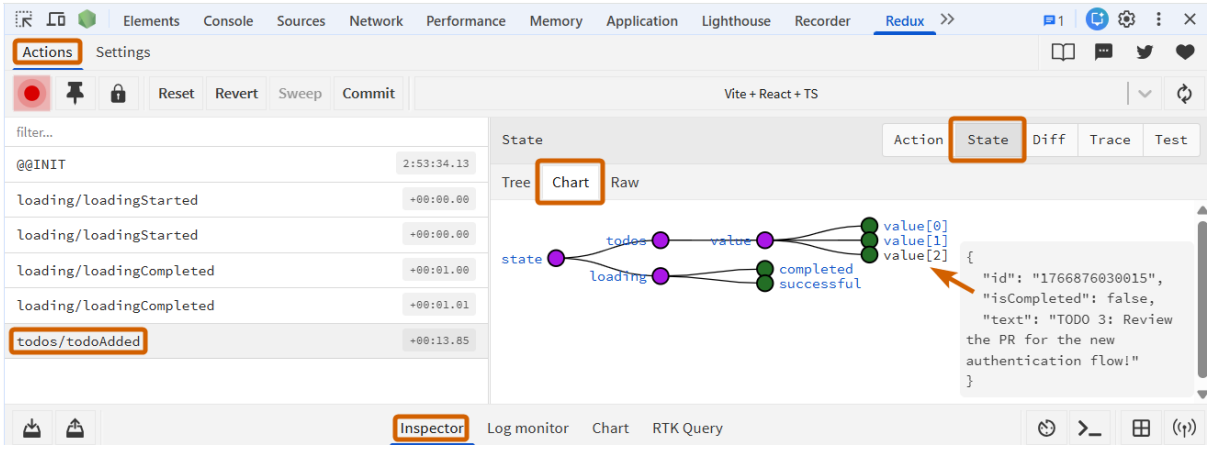
- **State:** Shows the current state of your application at the moment when the action was dispatched. In practice, this is the most useful debugging feature of Redux DevTools. It offers several views:
  - **Tree View:** An expandable, hierarchical view of your complete state object.



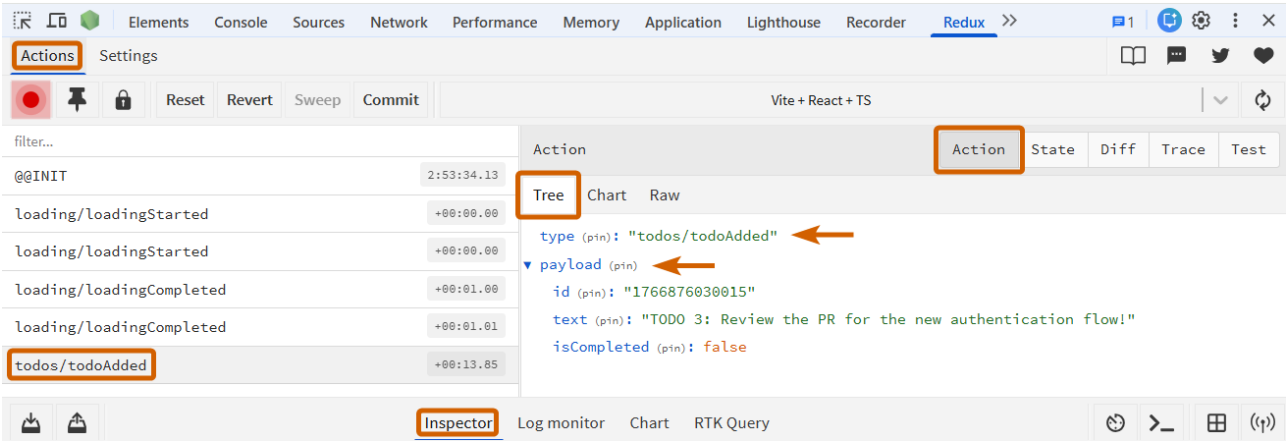
  - **Raw View:** A plain JSON representation of your state.



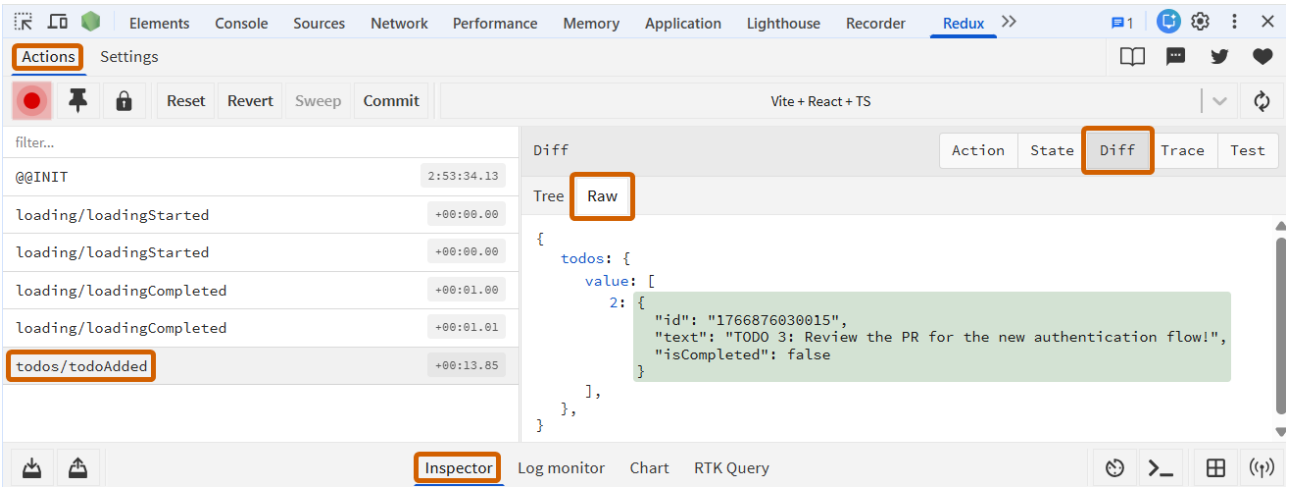
  - **Chart View:** A visual representation of your application's state tree. When the state becomes large or deeply nested, the chart view helps visualize relationships between slices.



- **Action:** Shows the dispatched action details: `type` and `payload` in different views: **Tree**, **Chart**, **Raw**.



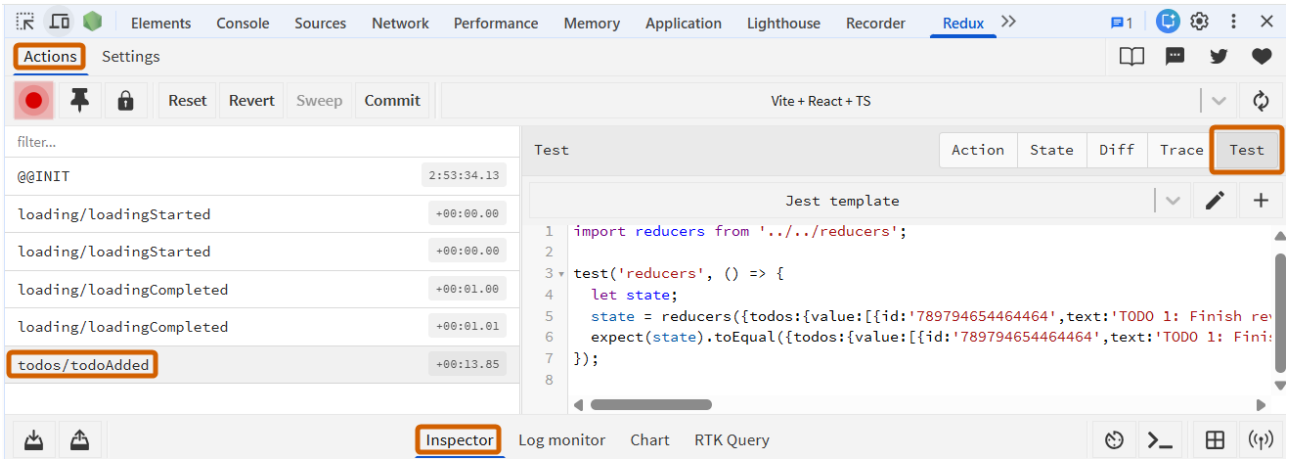
- **Diff:** Shows exactly what changed in the state as a result of the selected action, highlighting additions, edits and deletions. This is incredibly useful for debugging because reviewing the difference from the previous state helps you easily spot changes that led to a **buggy state**.



- **Trace:** Shows the trace in your source code where the action was dispatched, including the exact file path and exact line of code. You can click the trace and open the file in the **Sources** panel for further debugging. This helps you easily spot the source of the specific action causing the issue, especially when the application is complex with the same action being dispatched from different places.

- **Test:** Automatically generates a unit test template for the selected state change with different frameworks (e.g., **Jest**, **Mocha**). You can select a template for the desired framework and use it in your

test suite to verify state changes (typically via reducer), which significantly accelerates the testing workflow.

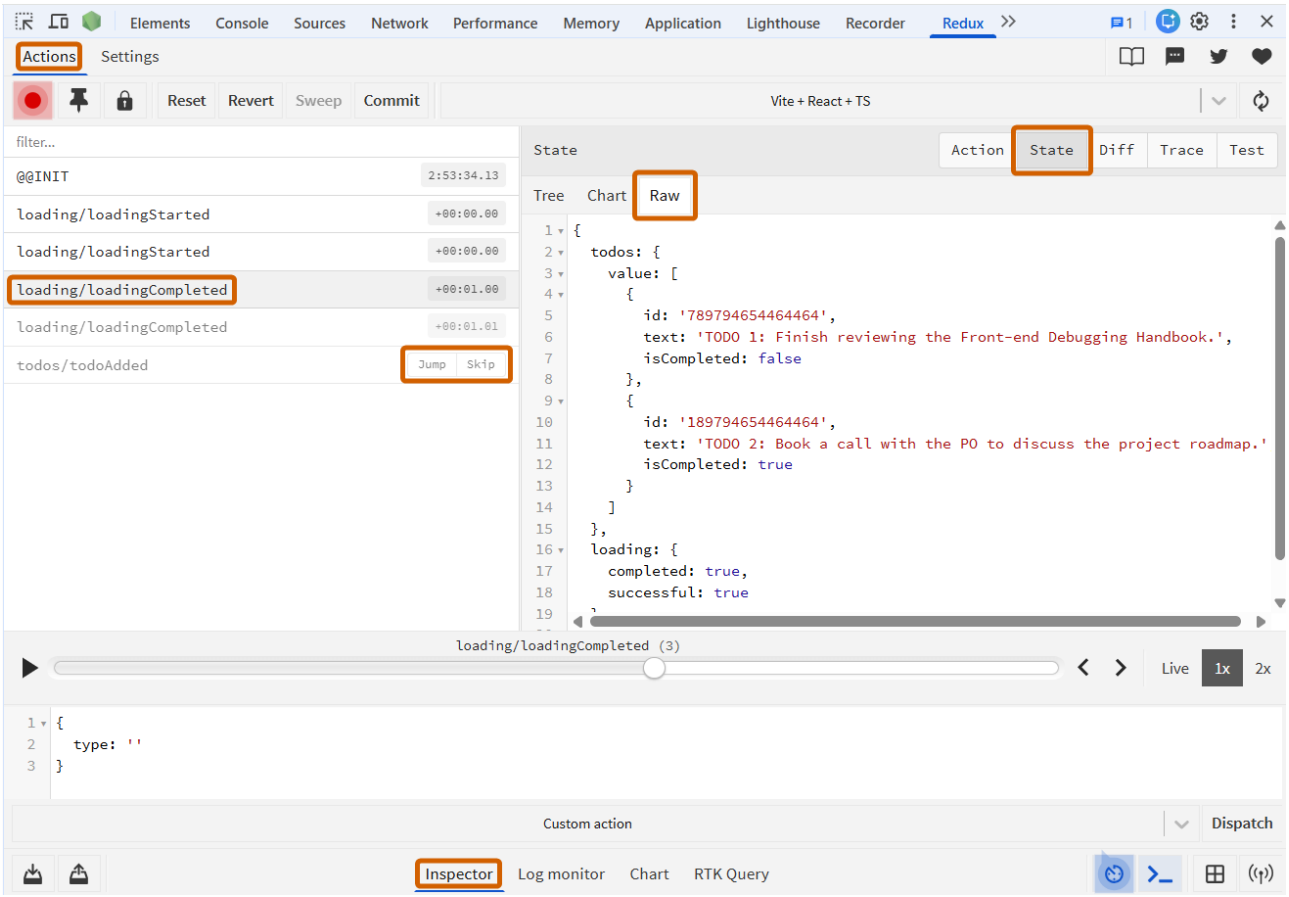


Redux DevTools: Time Travel Debugging

Time travel debugging is a core feature of the Redux DevTools extension, allowing you to **inspect**, **replay** and **skip** state changes that occurred in an application over time. The action log isn't just a log: it's a timeline you can interact with.

Below are the powerful features of time travel debugging:

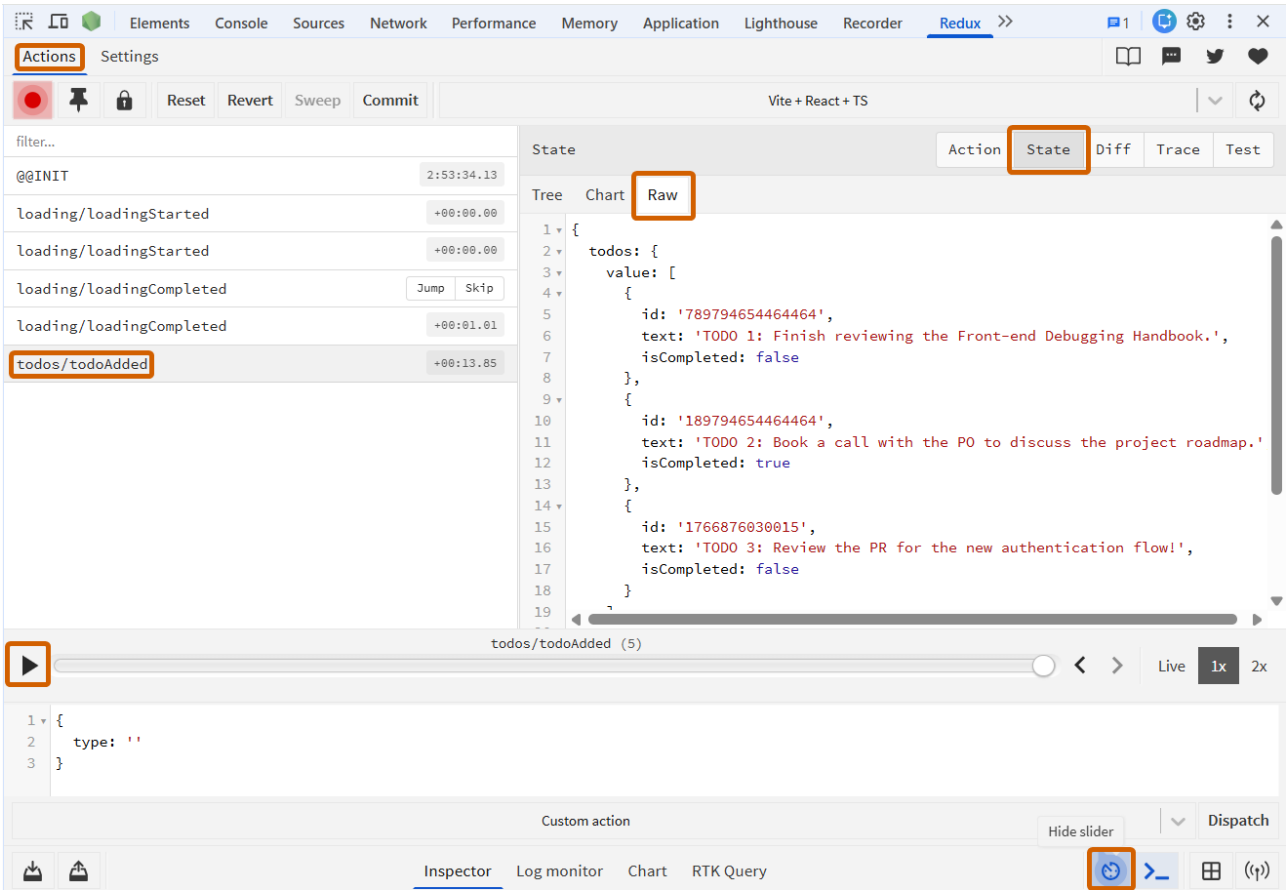
- **Jump or Skip Actions:**



You can **jump** directly to the state associated with an action or temporarily **skip** an action to disable it. This helps you test **what-if** scenarios, such as what would happen if a specific user action never occurred.

> **Note:** Hover over a specific action to view **Jump** and **Skip** operations.

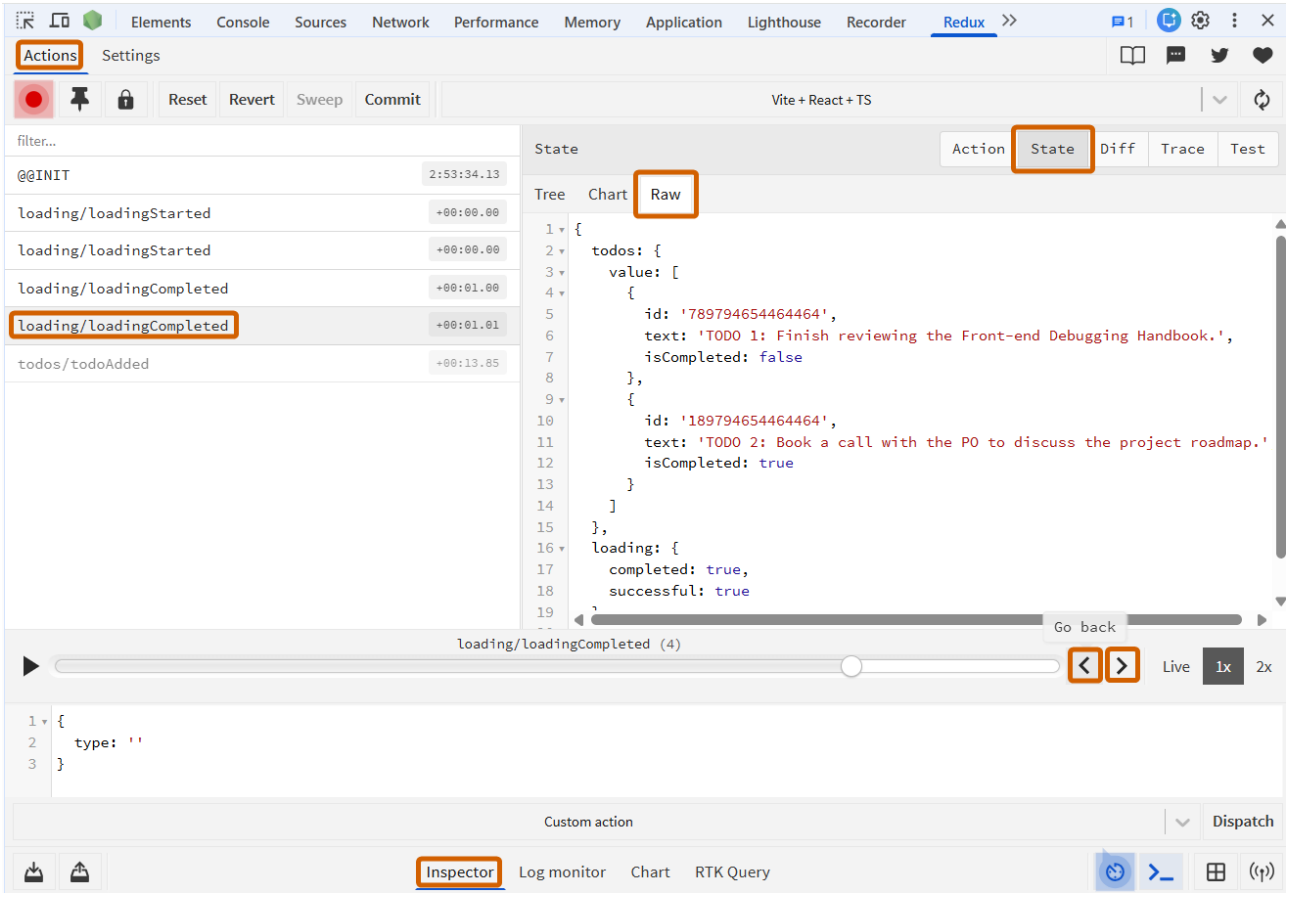
- **Replay Actions:**



You can use the time-travel slider to **play** (▶) and **pause** (⏸) through the actions one by one to see how your UI changes in response, making it easy to pinpoint exactly when and why a bug was introduced.

> **Note:** Use the **Show/hide slider** (🕒) icon in the **Monitors toolbar** at the bottom to show/hide the time-travel slider.

- **Go Back / Forward in Time:**



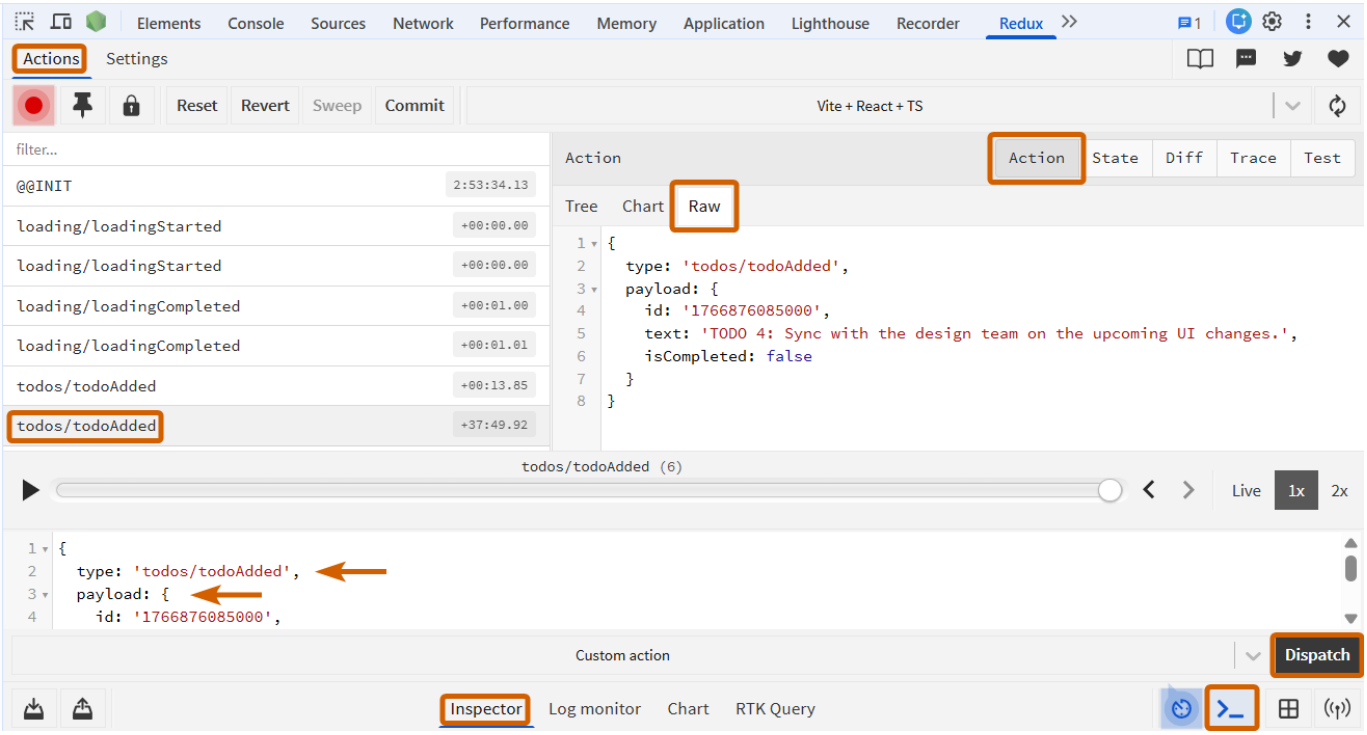
You can click the **Back** (❮) or **Forward** (❯) icons in the time-travel slider to move to the **previous** or **next** action in the log, reverting your application state to that moment. The UI will re-render to reflect that state.
The example above shows going back from the last state.

Redux DevTools: Dispatching a New Action

You can manually dispatch a new action with a custom type and payload directly from Redux DevTools. This is excellent for testing how your application state responds without having to trigger the action through the UI every time.

> **Note:** Use the **Show/hide dispatcher** (❯_) icon in the **Monitors toolbar** at the bottom to show/hide the action dispatcher.

For example, let's dispatch a new action to add a new todo:



Redux DevTools: Zustand Integration Example

While originally built for Redux (used with `react-redux`), Redux DevTools is now widely integrated with other frameworks (e.g., **NgRx** for **Angular**) and libraries which follow a similar, Flux-like pattern (e.g., **Zustand**). For primarily Redux-based libraries (e.g., **NgRx** for **Angular**), Redux DevTools is activated out of the box, whereas for others like **Zustand**, you need to integrate with middleware.

Let's explore how we can integrate Redux DevTools with a **Next.js React** application that has a **Zustand** store:

1. Integrate Redux DevTools

To enable Redux DevTools for your Zustand-based application, you need to import `devtools` from `zustand/middleware` and pass your state creator function to the devtools middleware, before passing the result to `create`. You can optionally configure settings provided by the `devtools` wrapper:

- `name`: Allows you to set a custom name for your store.
- `trace`: If `true`, the **Trace** tab in Redux DevTools Action/State explorer will be enabled.
- `enabled`: Used to conditionally enable or disable Redux DevTools integration in your application. For example, you can enable Redux DevTools only for lower environments (e.g., testing, staging) and disable it for production.

Below is an example of Redux DevTools integration in my POC Next.js application with Zustand store:

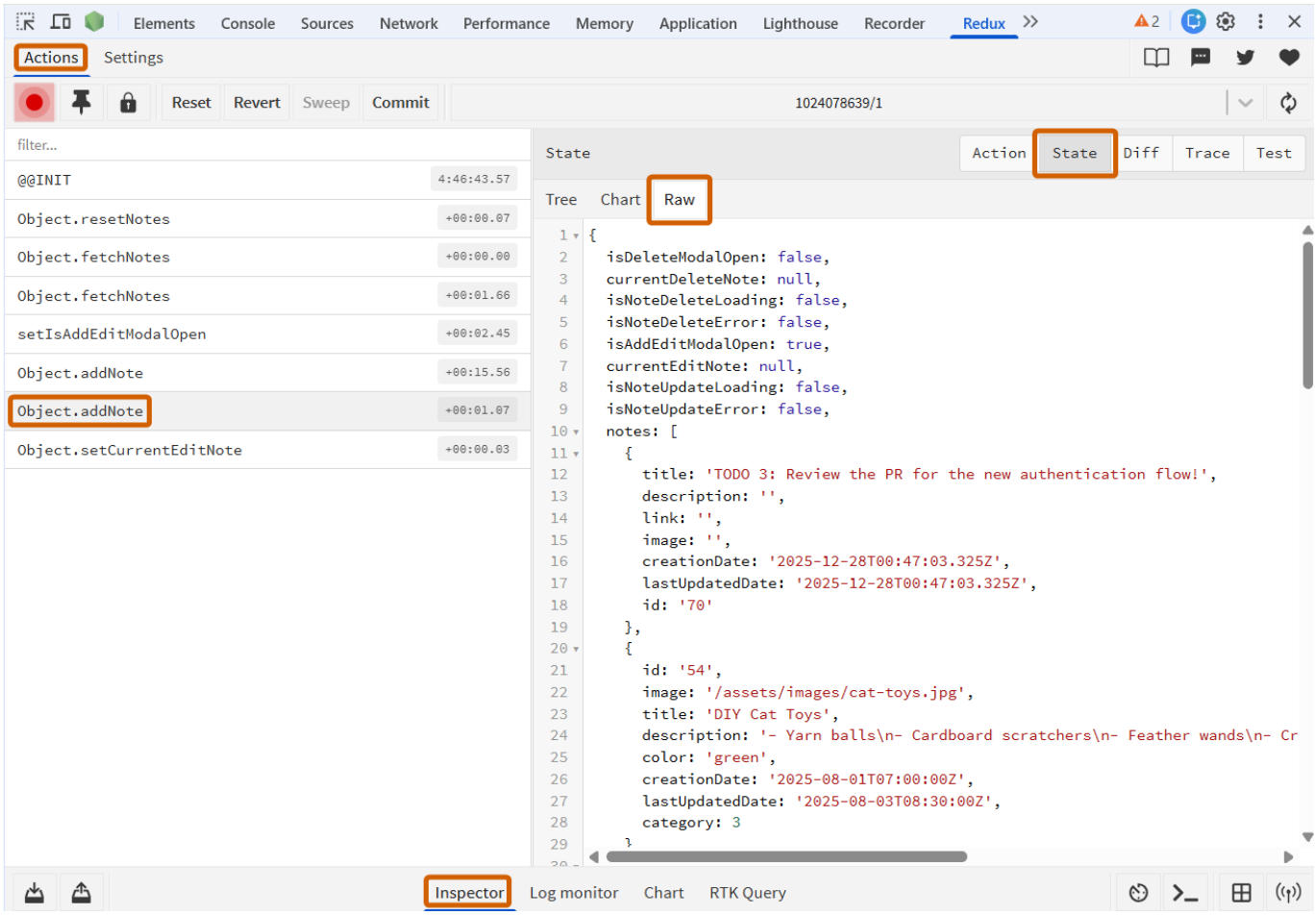
```
import { create, StoreApi } from 'zustand';
import { NotesStore } from '@/store/notes/notesStore.types';
import { devtools } from 'zustand/middleware';
```

```
export const useNotesStore = create(
  devtools((set: StoreApi<NotesStore>['setState']) => ({
    ...createDeleteNoteSlice(set),
    ...createAddEditNoteSlice(set),
    ...createNotesListSlice(set)
  }),
  {
    name: 'NotesStore', // Name of the store
    trace: true, // Enable Trace tab in Redux DevTools
    enabled: typeof window !== 'undefined' // Enable only on client-side
  }
  )
)
```

2. Debug in Redux DevTools

After this setup, you will see the **Redux** panel in your browser's DevTools and can leverage its main debugging capabilities.

Below is an example of what Redux DevTools looks like after adding a new note in the Next.js Note application:



While the core features are available, please note that **Dispatch custom action** is not available with this integration. This is simply because standard Zustand doesn't use reducers. Instead, actions are triggered

directly from the store functions. Technically, you could use the `redux` middleware from `zustand/middleware` and define custom reducers to fully mimic the Redux pattern (which would enable dispatching).
But I wouldn't advise doing this. The main advantage of Zustand is eliminating boilerplate code. The visualization and state tracking you get with this integration are already powerful. Adding complex reducers just to enable manual dispatching defeats the purpose of using Zustand.

Redux DevTools: Common Use Cases

The primary use of Redux DevTools is to facilitate manual testing and debugging during development.

- **View Actions and State**: View dispatched actions and the state changes associated with them. Inspect the action payload and the resulting state in various formats.
- **Time-Travel Debugging**: **Jump back** to previous states by skipping actions, effectively simulating an undo/redo history.
- **Dispatch Actions Manually**: Dispatch actions directly from the Redux DevTools interface to test how specific actions affect the store without writing application code.
- **Replicate Bugs in Production**: Export bug states history from a production environment into a `.json` file and import into a development environment to easily replicate bugs that are hard to catch otherwise.
  Note: **Export** ( ) and **Import** ( ) actions are located at the bottom-left part of the screen, in the **Monitors toolbar**.
- **Auto-generate Test Snapshots with Redux DevTools**: Dispatch custom actions and enhance your unit test suites with test case snapshots created by Redux DevTools based on the specific action change.
- **Export State Recording and Attach to Bug Ticket**: Export the recorded timeline of state changes representing a bug scenario and attach it in a bug ticket or send to teammates for further review.

5. PerformanceObserver Interface

With modern front-end technologies, developers are empowered with many tools to build performant applications. But to be sure that a web application is achieving its desired performance goals, developers need access to high-resolution performance measurement data.
This is where the `PerformanceObserver` interface becomes useful: it allows you to observe performance measurement events and be notified of new performance entries as soon as they are recorded in the browser's performance timeline.

PerformanceObserver Interface: Instance Methods

Member	Description
`po.observe()`	Specifies the set of entry types to observe. The performance observer's callback function will be invoked when a performance entry is recorded for one of the specified `entryType` values either via `type` or `entryTypes` options. **Important:** Use `{ buffered: true }` in the options to capture performance events that happened before the observer started (essential for LCP/FCP). This flag must be used only with the `type` option.
`po.disconnect()`	Stops the performance observer callback from receiving performance entries.
`po.takeRecords()`	Returns the current list of performance entries stored in the performance observer, emptying it out.

PerformanceObserver Interface: `entryType` Values

The read-only `entryType` property returns a string representing the type of performance metric. This is the value you pass to the `observe` instance method to start observing. All supported `entryType` values are available via the static `PerformanceObserver.supportedEntryTypes` property.

entryType	Description	Entry Instance Type
`element`	Reports the **load time of elements**.	`PerformanceElementTiming`
`event`	Reports **event latencies**.	`PerformanceEventTiming`
`first-input`	Reports the **First Input Delay** (FID).	`PerformanceEventTiming`
`largest-contentful-paint`	Reports the **largest paint** element triggered on screen.	`LargestContentfulPaint`

entryType	Description	Entry Instance Type
`layout-shift`	Reports the **layout stability** of web pages based on movements of the elements on the page.	`LayoutShift`
`long-animation-frame`	Reports instances of **long animation frames** (LoAFs).	`PerformanceLongAnimationFrameTiming`
`longtask`	Reports instances of **long tasks**.	`PerformanceLongTaskTiming`
`mark`	Reports your own **custom performance markers**.	`PerformanceMark`
`measure`	Reports your own **custom performance measures**.	`PerformanceMeasure`
`navigation`	Reports **document navigation timing**.	`PerformanceNavigationTiming`
`paint`	Reports **key moments of document rendering** (First Paint, First Contentful Paint) during page load.	`PerformancePaintTiming`
`resource`	Reports **timing information for resources** in a document.	`PerformanceResourceTiming`
`taskattribution`	Reports the **type of work that contributed significantly** to the long task.	`TaskAttributionTiming`
`visibility-state`	Reports the **timing of page visibility state changes** (foreground/background tab transitions).	`VisibilityStateEntry`

PerformanceObserver Interface: Example

Below are short steps on how to initialize a `PerformanceObserver` object:

1. First, initialize the `PerformanceObserver` object (`po`) using the constructor and provide a callback function which will perform the required actions when specific performance events are observed (e.g., highlighting the LCP element).
2. Next, start observing by calling `observe` on the `PerformanceObserver` object and passing the parameters of the metrics you want to observe (e.g., the `type` field as one of the `entryType` values listed above).

```
// This is a skeleton snippet, see full runnable example next

// 1. Initialize `PerformanceObserver` object
const po = new PerformanceObserver((list) => {
  let entries = list.getEntries();

  // Process entries

  const lastEntry = entries[entries.length - 1];
  console.log(`LCP is: ${Math.max(lastEntry.startTime - getActivationStart(), 0)}`);
});

// 2. Start observing
po.observe({ type: "largest-contentful-paint", buffered: true });

function getActivationStart() {
    // Get the page activation timestamp
}
```

Full Example: LCP Element Highlighting Code Snippet:

Let's explore an interesting example of a `PerformanceObserver` based code snippet that can be added as a **Snippet** in Chrome DevTools and help you monitor and highlight LCP candidate elements and the actual LCP element: Here is what it does:

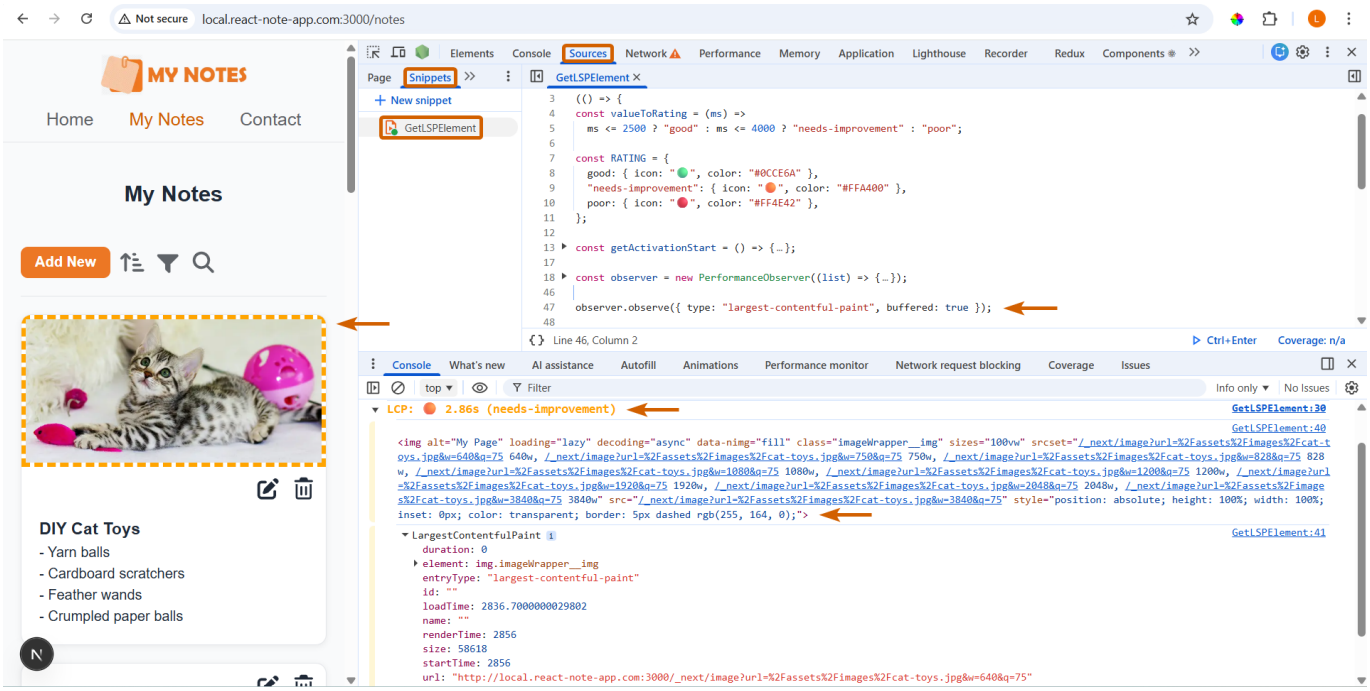
- Logs the **LCP** (**Largest Contentful Paint**) value and `LargestContentfulPaint` object in the **Console**.
- Adds a dotted line around the LCP element highlighting it in a color specific to its status:
  - `good`
  - `needs-improvement`
  - `poor`
- Observes and updates the LCP candidate as larger content is loaded (e.g., via lazy loading during scrolling).

> **Note:** This example is based on a snippet from webperf-snippets.nucliweb.net (License: **MIT 2026 © Joan León | @nucliweb**). I have modified it slightly to make it shorter for the handbook. You can find the full version on the original page, which also contains other useful performance snippet examples.

```
(() => {
    const valueToRating = (ms) =>
        ms <= 2500 ? "good" : ms <= 4000 ? "needs-improvement" : "poor";

    const RATING = {
        good: { icon: "🟢", color: "#0CCE6A" },
```

```
        "needs-improvement": { icon: "🟠", color: "#FFA400" },
        poor: { icon: "🔴", color: "#FF4E42" },
    };

    /**
     * Get the page activation timestamp
     * Relying on `activationStart` is important for cases
     * like prerender or bfcache restore,
     * where the page may exist before the user actually sees/uses it
     */
    const getActivationStart = () => {
        const navEntry = performance.getEntriesByType("navigation")[0];
        return navEntry?.activationStart || 0;
    };

    const observer = new PerformanceObserver((list) => {
        const entries = list.getEntries();
        // Last entry in the list is the LCP element
        const lastEntry = entries[entries.length - 1];

        if (!lastEntry) return;

        const activationStart = getActivationStart();
        const lcpTime = Math.max(0, lastEntry.startTime - activationStart);
        const rating = valueToRating(lcpTime);
        const { icon, color } = RATING[rating];
        const lcpDisplayTime = (lcpTime / 1000).toFixed(2);

        console.group(
            `%cLCP: ${icon} ${lcpDisplayTime}s (${rating})`,
            `color: ${color}; font-weight: bold; font-size: 14px;`
        );

        const element = lastEntry.element;
        if (element) {
            // Highlight LCP element
            element.style.border = `5px dashed ${color}`;
            console.log(lastEntry.element);
            console.dir(lastEntry);
        }

        console.groupEnd();
    });

    // Start observing LCP
    // (buffered:true includes earlier performance entries)
    observer.observe({ type: "largest-contentful-paint", buffered: true });
})();
```

Here is the result of running this snippet in the browser:



PerformanceObserver Interface: Browser Compatibility

- The `PerformanceObserver` interface and its methods are compatible with all major browsers.
- The `PerformanceObserver` interface is not compatible with **Deno** (a runtime environment for JavaScript, TypeScript and WebAssembly).
- The list of supported `entryTypes` varies per browser and is evolving. Always use the `PerformanceObserver.supportedEntryTypes` static property to check which are available in your current browser.
- Always check the most up-to-date detailed compatibility information in the MDN PerformanceObserver page.

PerformanceObserver Interface: Common Use Cases

- **Create browser snippets to monitor the performance of your web page**
  The idea of performance-based browser snippets is to allow you to monitor vital performance metrics as you explore the page and highlight associated elements.
  Below are examples of useful performance snippets based on the `PerformanceObserver` interface:
  - Display the CLS value when the focus of the browser is switched to another tab.
  - List the Largest Contentful Paint in the console and highlight the LCP element in the UI (the example we explored above).
  - Get a list of the BPP (bits per pixel) of all images loaded on the application.

  You can find these examples listed on webperf-snippets.nucliweb.net.

> **Note: BPP** (Bits Per Pixel) for an image indicates how much unique data is contained in each pixel on average and is used to measure **image entropy** (visual disorder and uncertainty).
>
> As of Chrome 112, LCP began to ignore images with very low content relative to their display size to prevent abuse by people cheating with fake "hero" images.
>
> - The **threshold** is currently `0.05` bits of image data per displayed pixel, and images that fall below this threshold will not be considered for LCP.
> - By ignoring very low-content images, Chrome will instead report the first paint with either text or a more contentful image as LCP. This may result in LCP times increasing.
> - By calculating BPP for your images, you can understand if they pass this threshold or are considered low content.

- **Collect essential performance data and send it to front-end monitoring platforms**
  You can collect key performance metrics directly from your application and send them to monitoring platforms (e.g., **Datadog**, **LogRocket**, **Sentry**).
  Common metrics collected through `PerformanceObserver` include:

  - **LCP (Largest Contentful Paint)** - measures how quickly the main content becomes visible.
  - **CLS (Cumulative Layout Shift)** - measures visual stability and unexpected layout movements.
  - **INP (Interaction to Next Paint)** - measures overall responsiveness to user interactions via the `event` entry type.
  - **Long Tasks** - detects tasks blocking the main thread for more than 50 ms.
  - **Long Animation Frames (LoAFs)** - detects slow animation frames that hurt smoothness.
  - **Resource Timing metrics** - measure network loading time for scripts, images, fonts, and other resources.
  - **Paint Timing (FCP/FP)** - measures initial render events such as First Paint and First Contentful Paint.

  These metrics can then be aggregated and visualized over time to detect regressions or performance degradation across deployments.

- **Create end-to-end integration tests to monitor performance metrics**
  An end-to-end testing example can be run in a CI/CD pipeline after each deployment to a testing environment to **monitor essential performance metrics and generate reports**.
  These tests compare the received data with established company thresholds and report if performance has decreased.

6. Performance: now() Method (High-precision Timing)

`performance.now()` is a Web API method that returns a high-resolution timestamp representing the number of milliseconds that have passed since the page navigation started in **Window** contexts, or the time when the worker is run in **Worker** and **ServiceWorker** contexts.

Performance: now() Method: Key Characteristics

- **High Precision:** The value returned from `performance.now()` is a floating-point number in milliseconds and offers up to microsecond-level precision. It is ideal for accurately measuring the duration of front-end operations, including very short ones.
- **Monotonic Clock:** The timestamp from `performance.now()` increases at a constant rate and is not affected by the system's clock. It **only moves forward**, providing a reliable and stable reference for performance measurement.

> **Note: Why Not Use `Date.now()` for Performance Measurement**
> You should avoid `Date.now()` for performance measurements because it is not designed for this purpose. Below are the main reasons:
>
> - **Precision:** `Date.now()` returns whole milliseconds (integers), whereas `performance.now()` returns floating-point numbers, offering up to microsecond-level precision.
> - **Reliability:** `Date.now()` is based on the system clock, which can be adjusted by the user or synchronized with a server, leading to inaccurate or even negative duration measurements. `performance.now()` is immune to these issues.

Performance: now() Method: Measuring Execution Time

The most common use for `performance.now()` is to measure the time a specific block of code takes to execute. By taking a timestamp before the start and after the end of the operation, you can calculate its precise duration.

For example, this may be useful if you want to create performance component tests and integrate them in a CI/CD pipeline:

- Mock the component function calls with large inputs to calculate how long execution takes.
- Then compare the results against accepted thresholds.
- Based on the results, you can either generate daily reports or fail the specific component testing step in the deployment pipeline when deploying to lower test environments.

```
// Get the starting timestamp
const startTime = performance.now();

// Perform a potentially slow operation
longRunningFunction();

// Get the ending timestamp
const endTime = performance.now();

// Calculate the duration
const duration = endTime - startTime;

console.log(`The longRunningFunction took ${duration.toFixed(2)} milliseconds to
execute.`);
```

Performance: now() Method: Simulating a Long Task (For Testing)

Another use case of `performance.now()` is to block the main JavaScript thread for **testing and demonstration purposes**.
For example, you may need to simulate a long-running synchronous calculation to see how it impacts your application's responsiveness or how your UI behaves when the main thread is frozen.

```
const start = performance.now();

while (performance.now() - start < 200) {
// This synchronous, blocking loop freezes the main JavaScript thread for 200ms.
}
```

> **Warning:** Intentionally blocking the main thread like this is an antipattern and should **never be used in production application code**. It will freeze the user interface and create a very poor user experience.

Performance: now() Method: Common Use Cases

- **Calculate the precise duration of your code block execution** by taking a timestamp before the start and after the finish of the operation.
- **Create performance component tests** that measure the performance of specific components, generate reports and include them in your CI/CD pipeline.

- **Simulate a long-running task** which blocks the main thread for testing and demonstration purposes (never use in production code).

7. Front-end Monitoring

In real life, your application may react differently on certain browsers or devices, or your end users may use some workflows which you cannot reproduce locally. Consequently, once the application is in production, unexpected errors and performance issues may occur, and teams need to quickly troubleshoot and debug to prevent a broken user experience.

Front-end Monitoring is the practice of recording errors, events and performance data from your application as it runs in a user's browser in real time. It is your primary tool for understanding and debugging issues that happen in a live **production** environment, which you often cannot reproduce locally. It is typically done through connecting and sending your application data to a specialized third-party application monitoring platform (e.g., **Datadog**, **LogRocket**, **Sentry**).

Each platform has its own pros and cons, from functionality to cost. Choosing the right tool really depends on your business needs and budget. However, application monitoring tools provide similar general functionalities:

- Search, filter and analyze real-time and historical logs to identify errors and find patterns.
- Get automated alerts for critical issues.
- View performance-related metrics in the dashboard via convenient UI charts.

Front-end Monitoring: Key Features

Below are the key features of front-end monitoring, including **reactive** and **proactive** monitoring aspects:

- **Real-time Monitoring and Error Detection:** Monitoring JavaScript errors, console logs and network requests in real time, which helps to identify issues that interrupt user interactions.
- **Real User Monitoring (RUM):** Capturing user interactions in an application and correlating those sessions with performance data such as Core Web Vitals, errors, resources and long tasks from the web or mobile application.
- **Synthetic Monitoring:** Creating no-code tests that simulate key user journeys with test users on the real production environment from different devices, browsers and locations. This helps ensure that important UX workflows work as expected and helps identify issues in the early stages, before they reach real users.
- **Session Replay:** Takes front-end monitoring a step further by offering visual replays of your users' browsing experience so that you can easily spot and reproduce any user errors.

From the features mentioned above, **Real-Time Monitoring and Error Detection**, **Real User Monitoring (RUM)** and **Session Replay** are considered **Reactive Monitoring** features because they are based on **capturing actual users'** interactions with your application.
In contrast, **Synthetic Monitoring** is a **Proactive Monitoring** feature because it is based on **simulating test users'** browser, mobile and API interactions in the real production environment.

Front-end Monitoring: Best Practices for Effective Logging

Logging data is one of the most important parts of the front-end monitoring process: it is the process of **defining and collecting the essential data** that will be used to monitor the health of your application. These best practices will help you log data effectively:

1. **Log only what is necessary:** In the majority of monitoring platforms, extra logs increase costs, so it is important to analyze the critical paths in your application and log them.
   For example, you may consider logging the following useful information:
   - Failed login/logout attempts.
   - Cart checkout steps.
   - Additional client-side information tied to each API failure, such as error code, page URL and page name.
   - Performance metrics such as **Core Web Vitals** (LCP, CLS, INP) using `PerformanceObserver` API. Some monitoring systems provide this functionality through RUM (Real User Monitoring) capabilities.
   - Execution duration of high-impact operations using `performance.now()` (e.g., how long a search filter or heavy calculation takes).
2. **Use Log Levels:** Categorize logs to make them filterable. This is a standard feature in all logging services.
   - `DEBUG`: Detailed information for development.
   - `INFO`: Normal application behavior (e.g., **User logged in**).
   - `WARN`: A potential issue that doesn't break functionality (e.g., **API response is slow**).
   - `ERROR`: A critical error that broke something (e.g., **Failed to process payment**).
3. **Provide Clear Messages and Context:** A log message without **context** is useless. Always include relevant data in the log, like the page name, current URL, user ID (non-sensitive) etc.
   For example, a log that says **Error** is not helpful. A log that says **Error: Failed to fetch user profile for userId: 123** is actionable.
4. **Protect User Privacy:** This is the most important rule. **Never log sensitive data.** Be extremely careful to exclude passwords, personal information (PII), credit card numbers, API keys or any other private data from your logs.

Front-end Monitoring: How to Integrate

Many teams have a detailed back-end logging and monitoring setup, but they are afraid to start with front-end monitoring because of the setup, maintenance overhead or cost. But the reality is that you can **start small**: just **forwarding your console logs** into the monitoring platform and **adding context** for your warning and error messages can be an easy, productive and fast way to start front-end monitoring. This is what we did for the financial legacy project.

How to Integrate: Error Detection Integration Example with Datadog

For the financial legacy project with micro-frontend and microservice architecture we used Datadog as a primary tool for application monitoring across the organization. While our back-end had a full setup with traces, production alerts, dashboards, metric charts, etc., for front-end monitoring we started small. Here is what we did:

- We only did basic integration setup inside micro-frontends with **Datadog Browser Logs SDK** which sends logs directly to Datadog from web browser pages.
- We forwarded console logs from our shell (parent application hosting micro-frontends) directly to the Datadog platform. In cases like this where traffic volume is high, it is crucial to follow limits to avoid unexpected costs. We forwarded only `warning` and `error` messages, so normally `info` messages were not visible in the production log explorer. `Info` messages were helpful for lower environment testing, but we had strict limits set by our Cloud Infrastructure team for lower environments in order not to burn the budget, both for front-end and back-end monitoring.
- The main custom log the micro-frontend implemented was logging API failures with `console.error` and adding extra context. For example, the same API may be called from different pages in the shell. By adding the **page name** as a context, we made it easy to find the location where the failure was coming from. This is basically what we had in the API failure log:
    - API failure code
    - Failure message
    - Context: `Upload Documents`, `Dashboard`, `Tax-slips`, `Statements`
    - Type: `Edit Document`, `Fetch Documents`, `Delete Document`

This simple setup and metadata made log filtering much easier. With Datadog rich filters, we could remove the noise, trim logs based on **type** and **context** (**page name**), and immediately list only errors coming from our micro-frontends, even when the same APIs were used across different micro-frontends.

This setup may not be your end goal, but it could be a good way to start with front-end monitoring in the early stages, particularly during a project MVP phase. Later, you can add other useful monitoring features, such as custom event logs (e.g., logout, login, payment failures), integrate Real User Monitoring (RUM), correlate RUM with back-end logs and traces for full-stack monitoring and add specific alerts.

How to Integrate: General Steps to Start with Integration

Different front-end monitoring platforms offer their specific nuances of integration. However, there are certain generic steps you usually take when integrating your front-end application with a monitoring platform:

1. Infrastructure Setup: There is typically a step where the Cloud Infrastructure team (or a staff front-end engineer) sets up the front-end monitoring integration into the shared company monitoring platform. As part of this setup, you are provided with connection credentials, such as **clientToken** and **site** (monitoring

platform URL of your organization) so that your front-end application can connect to the monitoring platform.

2. Connect to Monitoring Platform: Typically you perform this initialization only once by connecting to the monitoring platform in your root component.
Many big organizations even create shared libraries, so that you just install that library from `npm` and use a specific service to call it once to connect. This helps to keep monitoring platform logic in one unified place. Later, if the **URL** or the **key** is changed, or even if the vendor is changed, consumers will not need to modify their code: they can just update the library.

3. Create a Specific Logger Wrapper: It is a good practice to create a dedicated logger service (typically a singleton) for injecting custom log metadata. This allows you to encapsulate logging logic in a separate service and isolate it as a dedicated layer in your front-end application codebase. Below is a simple example I created for my Next.js POC project:

- Define types:

```
// logMessage.types.ts
export type LogMessageType = 'fetchNotes' | 'addNote' | 'deleteNote' |
'editNote' | 'sendMessage';
export type LogType = 'debug' | 'info' | 'warn' | 'error';
export type LogContext = 'dashboard' | 'myNotes' | 'myNotesLoadMore' |
'contact';
export type LogInfo = {
  type: LogType;
  messageType: LogMessageType,
  context: LogContext
}
```

- Define constants:

```
// logMessage.constants.ts
export const LogMessageTypeMapping: Record<LogMessageType, string> =
Object.freeze({
  'fetchNotes': 'Fetch Notes',
  'addNote': 'Add Note',
  'editNote': 'Edit Note',
  'deleteNote': 'Delete Note',
  'sendMessage': 'Send Message',
})

export const LogMessageContextMapping: Record<LogContext, string> =
Object.freeze({
  'dashboard': 'Dashboard',
  'myNotes': 'My Notes',
  'myNotesLoadMore': 'My Notes: Load More',
```

```
  'contact': 'Contact'
});
```

- Define Singleton Service for logging purposes (Datadog Example):

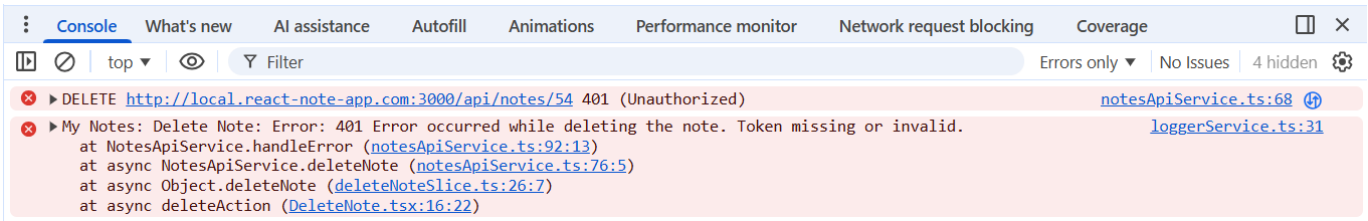
```
// loggerService.ts
import { datadogLogs } from '@datadog/browser-logs';

class LoggerService {
  private static instance: LoggerService;

  private constructor() {}

  /**
  * Logs message in the Console
  * And sends log message with specific context to monitoring platform
  */
  public log(
    logInfo: LogInfo,
    messageDetails?: string | null,
    error?: Error | null) {
    const type = LogMessageTypeMapping[logInfo.messageType];
    const context = LogMessageContextMapping[logInfo.context];
    const finalMessage = messageDetails
      ? `${context}: ${type}: ${messageDetails}:`
      : `${context}: ${type}:`;

    switch (logInfo.type) {
      case 'info':
        console.info(finalMessage);
        datadogLogs.logger.info(finalMessage, {context})
        break;
      case 'warning':
        console.warn(finalMessage);
        datadogLogs.logger.warn(finalMessage, {context})
        break;
      case 'error':
      default:
        if(error) {
          console.error(finalMessage, error);
          datadogLogs.logger.error(finalMessage, {context}, error);
        } else {
         console.error(finalMessage);
         datadogLogs.logger.error(finalMessage, {context});
        }
      break;
    }
  }
```

```
  public static getInstance() {
    if(!this.instance) {
      this.instance = new LoggerService();
    }

    return this.instance;
  }
}

const loggerService = LoggerService.getInstance();

export default loggerService;
```

4. Integrate Logger Wrapper into Application Logic: The last step is to use the service to call its log method in actual API calls:

```
// Error message should be constructed from error catch block
try {
  // Your API logic here...
  await notesApiService.deleteNote(noteId);
}
catch(error: unknown) {
  loggerService.log({
    type: 'error',
    context: 'myNotes',
    messageType: 'deleteNote'
  }, null, error as Error);
}
```

The Console output below displays the structured data sent to the monitoring platform:



Front-end Monitoring: Useful Tips

1. **Source Maps:** In order to make your front-end application's size smaller, normally its code is minified when it is built for release. That's why to link errors to your actual code, often you need to upload the source map files of your build to your monitoring vendor platform.
   To make this automatic, you should configure your build pipeline to automatically upload source

maps to your monitoring platform. This helps the front-end monitoring platform to link error stacks to actual source code and helps you debug issues at a more granular level.

2. **Filtering Noise and Costs:** If you decide to redirect all your console logs to the monitoring platform, you need to keep in mind that you may have errors from third-party resources, so you may need to make sure they are trimmed out before going to the monitoring platform.
   Many monitoring platforms provide a callback in SDKs (e.g., `beforeSend` for Datadog **Browser Logs SDK**), which you can use to explore every log before sending it to the monitoring platform. This way you can trim and narrow down the logs sent to the platform to exclude third-party vendor logs.
   This method not only helps to reduce noise but is also useful for cost saving, because less traffic on the monitoring platform means lower costs.

Front-end Monitoring: Common Use Cases

Front-end monitoring is a critical practice for maintaining application health and ensuring long-term productivity. It provides visibility into how your application behaves in the real world, on a user's specific device. This is crucial for:

- **Debugging Environment-specific Issues**: Users have countless combinations of devices, operating systems, browsers and network speeds. Front-end monitoring captures errors and context from these diverse environments, revealing bugs that you may never see on your own development machine or testing environments.
- **Investigating Non-reproducible Bugs**: A user's specific actions or data can create unique edge cases. When a user reports a bug that you can't replicate, logs and session replays often provide the step-by-step history needed to understand and fix the problem.
- **Proactive Monitoring and Alerting**: By sending logs to a centralized platform and creating synthetic monitoring test cases, you can monitor the overall health of your application and get alerted when critical production issues happen. This helps you fix bugs before they impact a large number of users.
- **Performance Monitoring**: By enabling Real User Monitoring (RUM), you can get an overview of how your application behaves on real user devices and track Core Web Vitals (LCP, CLS, INP) in real time. This helps you to identify performance bottlenecks and ship improvements early, before users even notice.

8. Chrome DevTools MCP Server (Public Preview)

Google launched the public preview of the **Chrome DevTools Model Context Protocol (MCP) server*** in September 2025, bringing the power of Chrome DevTools directly to AI coding assistants.

It addresses a fundamental limitation of AI coding agents: they cannot naturally observe how the code they generate behaves inside a real browser. Without visibility into runtime behavior, they operate with limited context.

The Chrome DevTools MCP server changes this by letting your **coding agent** (such as Gemini, Claude, Cursor or Copilot) **control and inspect a live Chrome browser**. This gives your AI coding assistant access to the full power of Chrome DevTools for reliable automation, in-depth debugging and performance analysis, significantly improving accuracy when identifying and fixing issues.

> **Important Note:** The Chrome DevTools MCP Server is currently in **Public Preview** (versions `0.x.x`).
>
> - Preview mode is intended for local debugging, development and experimentation only. It is not intended for production usage yet, as breaking changes may occur without major version updates.
>   Please check the official reference at github.com/ChromeDevTools/chrome-devtools-mcp by the time you read this handbook, as it may already be in a stable release and safe for production use.
> - To use it, you need **Node.js** `v20.19` or **any newer LTS version** and the **latest stable Chrome** browser installed.

Chrome DevTools MCP Server: Terminology

Below are short explanations for terms that you will encounter across this section which are marked with *:

- **MCP server:** Program that exposes specific capabilities to AI applications through standardized MCP (Model Context Protocol) interfaces.
- **HAR file (HTTP Archive):** A JSON-formatted log of a web browser's interactions with a website. It records all browser network activity: every request, response, header and sometimes even the response body. Because it can contain sensitive data such as API keys, cookies or user information, always treat HAR files as confidential.
  To learn how to import/export a HAR file in the Chrome browser, check the Network Panel: Downloading and Importing HAR (HTTP Archive) Files section.
- **Puppeteer:** A JavaScript library which provides a high-level API to automate both Chrome and Firefox over the **Chrome DevTools Protocol** and **WebDriver BiDi**. It can be used to automate anything in the browser, from taking screenshots and generating PDFs to navigating and testing complex UIs and analyzing performance.

Chrome DevTools MCP Server: Installing in Cursor IDE

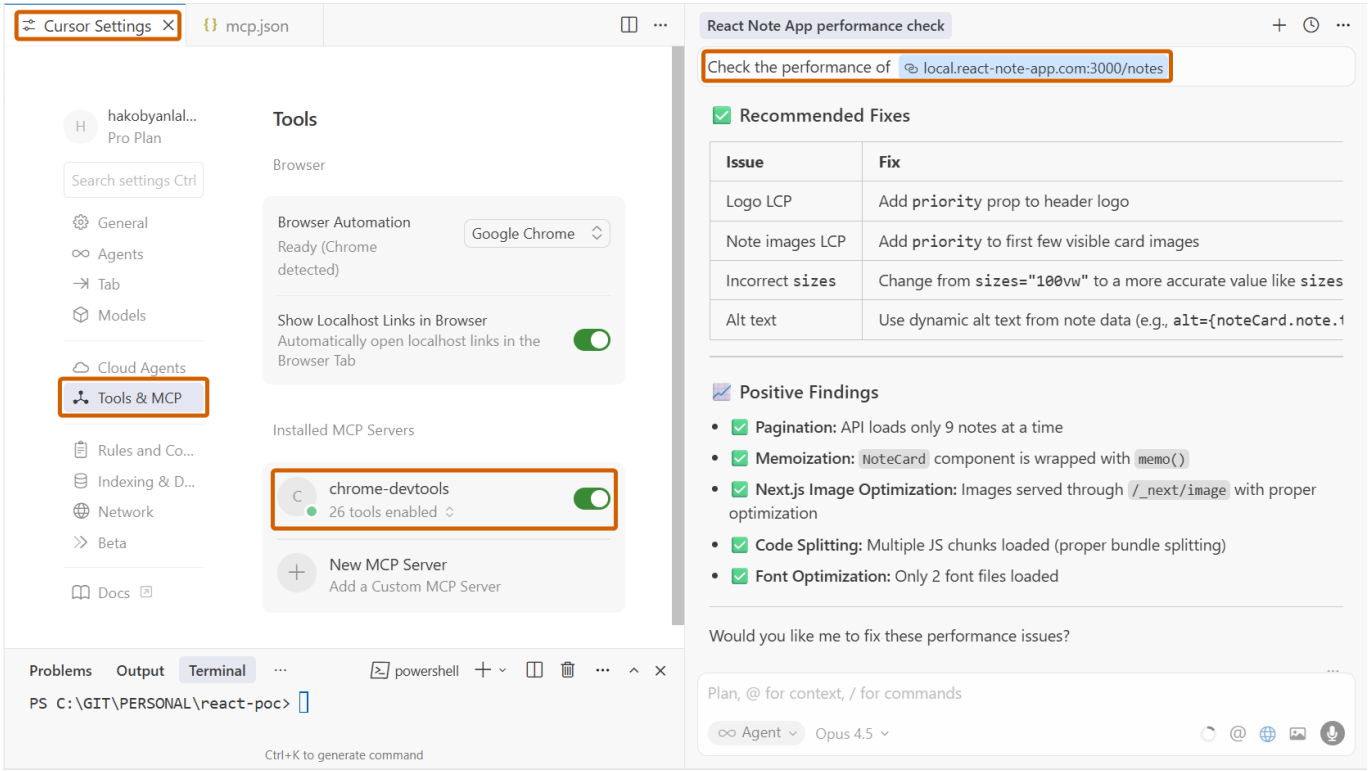
Below are the steps to install Chrome DevTools MCP Server in Cursor IDE:

1. Go to **File > Preferences > Cursor Settings**
2. Choose **Tools & MCP > New MCP server**
   It opens the `mcp.json` file where you can add your MCP server config:

   ```
   {
     "mcpServers": {
       "chrome-devtools": {
         "command": "npx",
         "args": [
           "-y",
           "chrome-devtools-mcp@latest",
           "--no-usage-statistics",
           "--isolated=true"
         ]
       }
     }
   }
   ```

   You can pass various options via your MCP server JSON configuration in the `args` property.
   For the full list, check the Configuration Options section.
3. After that, you can start sending an MCP-related prompt to your MCP client.
   For example, write the following prompt in Cursor IDE Agent chat:

   ```
   Check the performance of http://local.react-note-app.com:3000/notes.
   ```
4. When the prompt is first executed, make sure to add **chrome-devtools** commands to **MCP Allowlist** so the coding agent doesn't ask for permission each time it runs a command.
5. During the code execution the coding agent will open the Chrome browser and execute necessary actions there. If you didn't specify `--headless=true` in `args`, you will see the browser UI opened and controlled by the coding agent. In case of headless mode, the browser actions will be performed under the hood.
6. Once the analysis is complete, a full report will be displayed with actionable suggestions for improvements.



Chrome DevTools MCP Server: Key Features

- **Get Performance Insights:** Uses Chrome DevTools to record traces and extract actionable performance insights.
- **Advanced Browser Debugging:** Analyzes network requests, takes screenshots and checks the browser Console via Chrome DevTools.
- **Reliable Automation:** Uses **Puppeteer*** to automate actions in Chrome and automatically wait for action results.

Chrome DevTools MCP Server: Configuration Options

The Chrome DevTools MCP server supports the following configuration options which you can pass via your MCP server JSON configuration in the `args` property:

- `--browserUrl, -u`: Connect to a running Chrome instance using port forwarding.
  - **Type:** `string`
- `--headless`: Whether to run in headless (no UI) mode.
  - **Type:** `boolean`
  - **Default:** `false`
- `--executablePath, -e`: Path to a custom Chrome executable.
  - **Type:** `string`

- `--isolated`: If specified, creates a temporary user-data-dir that is automatically cleaned up after the browser is closed.
    - **Type:** `boolean`
    - **Default:** `false`
- `--channel`: Specify a different Chrome channel that should be used. The default is the stable channel version.
    - **Type:** `string`
    - **Choices:** `stable`, `canary`, `beta`, `dev`
- `--logFile`: Path to a file to write debug logs to. Set the env variable `DEBUG` to `*` to enable verbose logs. Useful for submitting bug reports.
    - **Type:** `string`
- `--viewport`: Initial viewport size for the Chrome instances started by the server. For example, `1280x720`. In headless mode, the max size is 3840x2160px.
    - **Type:** `string`
- `--proxyServer`: Proxy server configuration for Chrome passed as `--proxy-server` when launching the browser.
    - **Type:** `string`
- `--acceptInsecureCerts`: If enabled, ignores errors relative to self-signed and expired certificates. Use with caution.
    - **Type:** `boolean`
- `--chromeArg`: Additional arguments for Chrome. Only applies when Chrome is launched by `chrome-devtools-mcp`.
    - **Type:** `array`

You can also run `npx chrome-devtools-mcp@latest --help` to see all available configuration options.

Chrome DevTools MCP Server: Cursor Workflow: AI-assisted E2E Test Case with Report Generation

Let's explore a Proof-of-Concept (POC) example of a test case written in plain language for the Cursor coding agent. With zero hand-written code, the AI agent can automatically perform the test and generate a report. This is an example of one specific test case, but you can use it as a template for generating tests for a specific list (`.json` file).

The main steps it performs are:

- Navigates to the given URL (opening a Google Chrome browser).
- Finds the specified note element.
- Clicks its edit icon to open the modal.
- Edits the title.
- Saves the changes.
- Validates that the changes were saved correctly.
- Collects console logs and network requests throughout the process and generates a final report.

After experimenting with different AI models in **Cursor IDE** and trying the latest version of **Chrome DevTools MCP**, I found out that it works best with Anthropic's latest models and this simplified prompt:

```
Open @http://local.react-note-app.com:3000/notes

Here are the Important Rules that you need to follow:
- Execute exactly the listed actions. Do not click any other element than the one
specified.
- Use the exact selectors below. If a selector returns 0 or more than 1 element, stop
and return an error with a 300-char DOM excerpt.
- Wait for network idle before the first action, and for the UI to be stable (no
layout/size changes) for 50 ms before each click or type.
- ALWAYS use Chrome DevTools commands (mcp_chrome-devtools_*) instead of JavaScript
evaluation for ALL browser interactions (query, click, fill, etc.).

Step 1 - Open and start capturing:
- Wait for network idle.
- Begin capturing console logs (fields: ts, level, text, url, line, column) and
network request/response information.
- Generate a timestamp in format `DD-MM-YYYY-HH-mm` (e.g. `15-10-2025-00-12`) and
store it as `RUN_DATE`.

Step 2 - Click ONLY the edit icon for the target note:
- Resolve a SINGLE node via selector '[data-id="54"] .svg-link:first-child'
- Do not take snapshots to verify, do not try to find the element again, do not
question the UID.
- Click immediately the found node ONCE. Do not click any other elements with class
"svg-link".

Step 3 - Wait for modal and edit the title:
- Wait ONLY for 50ms until a dialog is present with this selector: '#modal'
- Within the dialog, locate the title input using the first that exists with this
selector: 'input[name="title"]'
- Ensure the input is interactable and stable for 50 ms.
- Focus the input, select all (Ctrl/Cmd + A), delete, then type EXACTLY: DIY Cat Toys
Test1.

Step 4 - Save and close:
- Locate the save button inside the dialog using the first that exists with this
selector: 'button[name="submitButton"]'
```

```
- Ensure the button is visible and stable for 50 ms, then click it ONCE.
- Wait until the dialog disappears (the dialog selectors from Step 3 should not
exist).

Step 5 - Verify the update:
- On the list view, confirm that the element with the '[data-id="54"] h3' now contains
text "DIY Cat Toys Test1".
- If not, return failure with a 300-char DOM excerpt around [data-id="54"].

Step 6 - Summarize console and write report:
- Group captured console logs into Errors, Warnings, Info (info + debug).
- Group request and response information of the API requests.
- Redact secrets using patterns: `["apiKey=\\S+", "Authorization: Bearer \\S+",
"token=\\S+"]`
- Provide counts per group and up to 2 example lines per group (after redaction).
- Create Markdown at path: `apps/note-app-nextjs/e2e/report-${RUN_DATE}.md`
  Use this EXACT template:

# Note Edit E2E Report — ${RUN_DATE}
- **Date:** ${RUN_DATE}
- **URL:** http://local.react-note-app.com:3000/notes
- **Edited Note ID:** 54
- **Old Title:** DIY Cat Toys
- **New Title:** DIY Cat Toys Test1
- **Result:** {Success | Failed + short reason}

## Console Summary
| Type    | Count | Examples |
|---------|-------|----------|
| Errors  | {E}   | {up to 2 redacted examples} |
| Warnings| {W}   | {up to 2 redacted examples} |
| Info    | {I}   | {up to 2 redacted examples} |

## Observations
- {1-3 concise bullets on issues/patterns}

## Suggested Fixes
- [ ] {short, actionable suggestion}
- [ ] {optional second suggestion}

- Verify the file exists and is > 200 bytes, then return a final JSON object:
  { "status": "ok" | "failed", "noteId": "54", "newTitle": "DIY Cat Toys Test1",
   "reportPath": "apps/note-app-nextjs/e2e/report-${RUN_DATE}.md",
   "counts": { "errors": E, "warnings": W, "info": I } }
```

Important Notes: Challenges / Downsides

- This prompt was tested in the Cursor IDE, and the `Rules` section proved to be critical for ensuring the AI agent executed the test correctly.

Specifically, without the instruction to `ALWAYS use Chrome DevTools commands (mcp_chrome-devtools_*)`, the AI agent would fall back to using direct JavaScript commands to modify data. This resulted in non-realistic interactions because it bypassed the actual user event simulation that the Chrome DevTools MCP commands provide.

- The AI agent was not always stable and sometimes ignored rules in the prompt. So besides adding extra instructions to reinforce rule-following, choosing the right model was also essential for achieving stability.
  Choosing a dedicated model by **Anthropic** (e.g., **claude-4.5-sonnet**, **Sonnet 4.5**, **Opus 4.5**) gave the best experience among all the tested models. However, since different models have different token costs, with more powerful ones being pricier, it is important to **keep cost in mind before choosing the model** to work with Chrome DevTools MCP.

> **Note:** These additional prompt instructions and stability issues are expected because the Chrome DevTools MCP Server is currently in **Public Preview**. Preview mode is not fully stable, and certain behaviors may require extra guidance in your prompts.
> Please check the official reference by the time you read this handbook, as the MCP server may already be in a stable release, allowing you to use a much more simplified prompt and have a seamless experience.

Chrome DevTools MCP Server: Cursor Workflow: Configuring AI Agents for Self-testing

Having your AI coding agent setup according to best practices is essential to achieve effective debugging workflows.
The productive setup in your AI coding agent assumes you specify a dedicated folder to store rules, skills and workflows to tell the AI coding agent to follow them during feature planning and implementation. Those could be general agent behavior rules, coding standards, specific domain skill knowledge, code review or security audit workflows.

- This setup is an important step because without it, the AI coding agent may not generate code according to accepted standards and quality for your project.
- This practice not only helps during your local development, but also helps to create a shared knowledge base and practices among team members as you commit this folder into the repository.

In this section, we will explore an example of a productive workflow in **Cursor** coding agent, following best practices.

Set Up `.cursor` Folder with Rules and Commands

Below is a simple example of how you can organize productive setup of rules, workflows (commands), skills and feature specs in your Cursor IDE:

```
# .cursor Folder Structure

.cursor/
├── commands/
│   ├── code-review-checklist.md
│   ├── development-self-testing.md
│   └── security-audit.md
├── features/
│   ├── create-settings-page.md
│   └── note-edit-e2e-test.md
├── rules/
│   ├── agent-behavior.mdc
│   └── standards/
│       └── nextjs-react-typescript-best-practices.mdc
└── skills/
    └── accessibility/
        └── SKILL.md
```

Rules: These are general rules that should be applied by the coding agent across the project. Cursor by default looks into the `rules` folder to understand your rules and include them in the Agent chat context.

- You can specify **rules** to **always apply** like **agent behavior** rules, or to apply **intelligently**, such as **Next.js, React and TypeScript coding standards** applied to `.ts` and `.tsx` files only.
- As per Cursor standard, **rules** are preferable to follow `.mdc` format which is a markdown file with frontmatter metadata (e.g., `globs`, `alwaysApply`) and content.
- Example of `agent-behavior.mdc`:

```
---
globs:
alwaysApply: true
---

# Agent Behavior

- Each time when a specific task is given, don't overstep from your role. Always
ask to confirm next steps before processing.
- During the planning stage always check rules and never modify any files.
```

Commands: These are reusable workflows that you define and apply when needed, such as **development self-testing** or **security audit** workflows. Unlike **rules**, **commands** are not automatically applied by the coding agent. After specifying your workflows in the specific `commands` folder, you can select them by typing `/command-name` in the Agent chat and ask the AI agent to execute them.

Skills: Teaches agents how to perform domain-specific tasks by extending AI agents with specialized capabilities.

- For example, you can specify accessibility guidelines and help AI agent to broaden its skills to specific accessibility domain.
- **Skills** can also be manually selected by typing `/skill-name` in the Agent chat, similar to **commands**.
- By default, **skills** are automatically applied when the AI agent determines they are relevant. Set `disable-model-invocation: true` to make a **skill** behave like a traditional slash command, where it is only included in context when you explicitly type `/skill-name` in chat.
- Each **skill** should be a folder containing a `SKILL.md` file. A **skill** is defined in the `SKILL.md` file with YAML frontmatter:
- Example of **accessibility skill** (`accessibility > SKILL.md`):

```
---
name: accessibility
description: Audit and improve web accessibility following WCAG 2.1
guidelines. Use when asked to "improve accessibility", "a11y audit",
"WCAG compliance", "screen reader support", "keyboard navigation" or
"make accessible".
---

# Accessibility (a11y)

Your guideline here...
```

Features (Specs): This is a custom folder which you specify to store your feature implementation specs.

- After defining a feature spec, you can drag/drop it to your Agent's chat to execute, instead of writing a big prompt inline.
- A saved feature spec helps to reduce efforts of writing spec each time from scratch for you and your team.

You can review all **rules**, **commands** and **skills** in **Cursor Settings > Rules, Skills, Subagents** section.

Set Up Development Self-testing Workflow

1. Create Development Self-testing Workflow Spec File

Below is a short example of `development-self-testing.md` workflow file under `.cursor/commands` folder, which tells the AI agent to verify its implemented features after completion:

```
# Development Self-testing

- After you finish implementing the task, open the URL http://local.react-note-
app.com:3000 in Chrome and test the implemented feature in the UI.
- ALWAYS use Chrome DevTools commands (mcp_chrome-devtools_*) instead of JavaScript
evaluation for ALL browser interactions (query, click, fill, etc.).
```

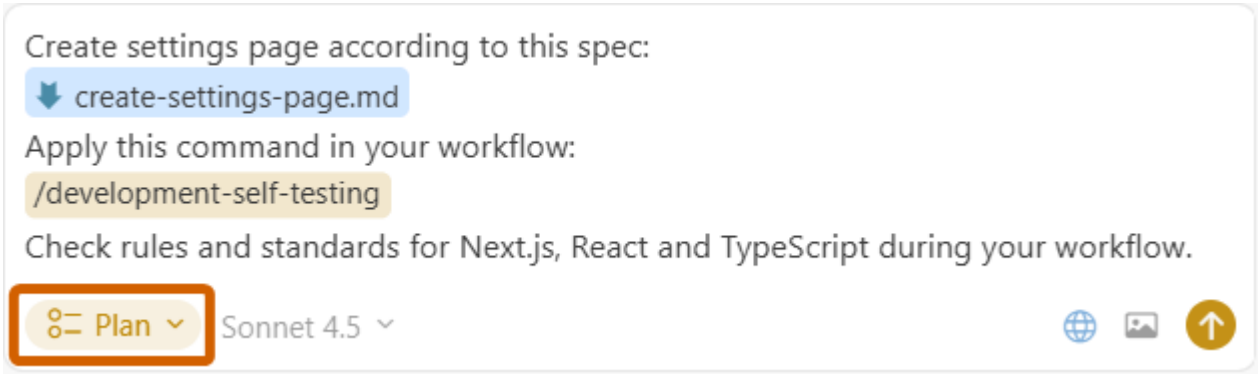
2. Create New Feature Spec File

Create the new feature prompt in a `create-settings-page.md` file in the `.cursor/features` folder to describe the workflow of building the feature. Specify any coding standards or examples the AI agent should use to build the feature.

```
# Create Settings Page

Add a new feature: a new route named "settings".
Add it to the header as a link. It should have the following data:
- Page Title: Settings
- Form Fields:
    - First Name (text input)
    - Last Name (text input)
    - Email (email input)
    - Items Per Page (number input)
    - Enable Email Sync (checkbox)
    - Buttons: Save, Cancel
- Action: For now, just log all form values to the Console when clicking Save.
```

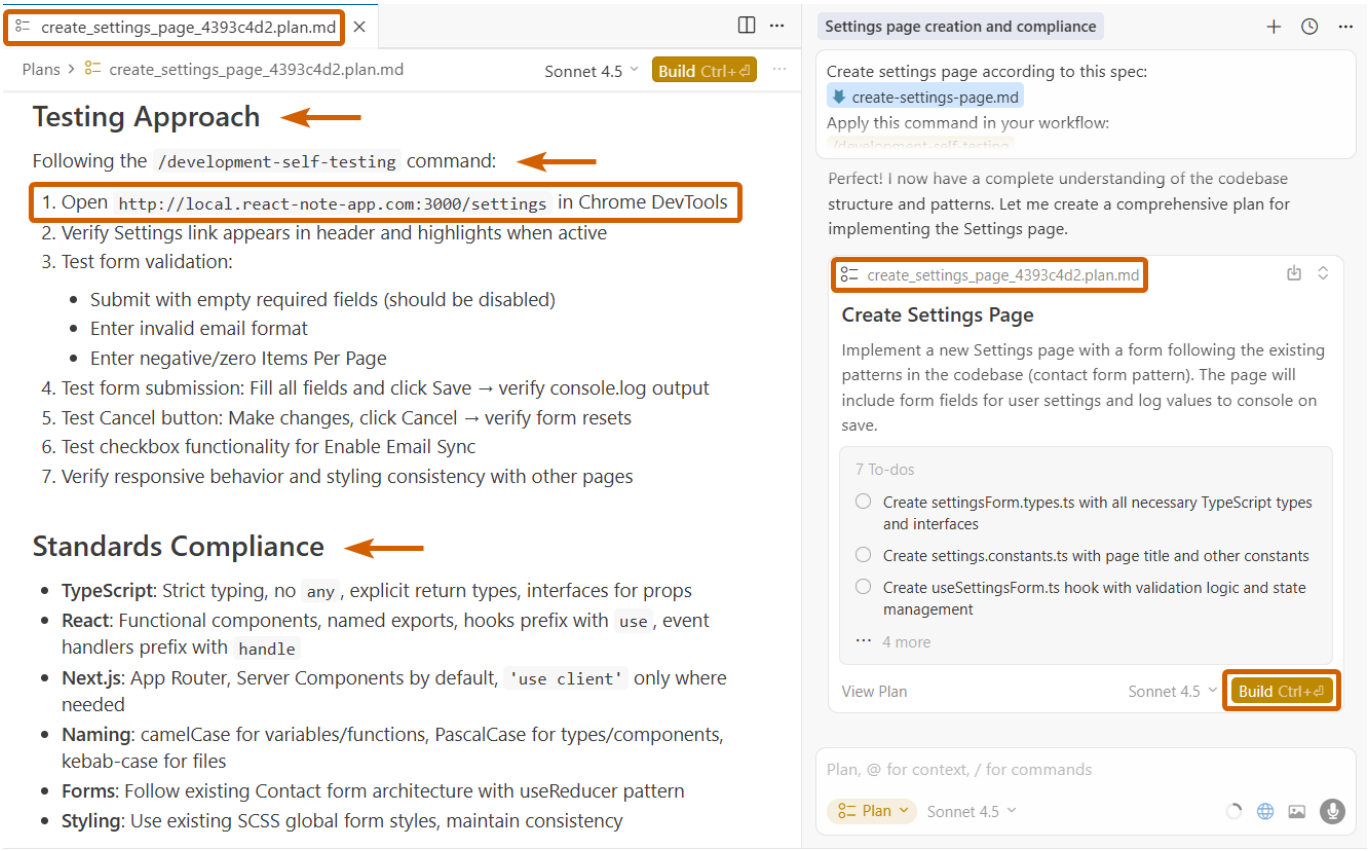
3. Plan Feature

- Drag/drop the feature creation guide file (e.g., `create-settings-page.md`) to the Agent chat, select the `development-self-testing.md` command and use **Plan** mode to start planning the feature. You can also mention any best practices that you want the AI agent to follow.
- Planning the feature helps the AI agent understand your codebase and rules and make decisions (plan) about building the feature rather than just proceeding with its blind understanding. It gives you the opportunity to review the plan and guide the AI agent to produce better quality and fewer bugs.
- For the planning mode you may consider choosing an advanced thinking model with bigger context window for better outcome. As mentioned earlier, **Anthropic** models perform pretty well for these tasks.



4. Build Feature

Once finished, you can review the plan, change it or approve the plan by pressing **Build** and it will start generating the final feature for you.



5. Test New Feature

After execution is finished, the AI agent will open the URL you specified in the `development-self-testing.md` command file and will test the implemented feature.

Chrome DevTools MCP Server: Security Considerations

Chrome DevTools MCP is an **official integration from Google**. However, because DevTools MCP can see everything in an open browser tab, caution is required when your AI client also connects to **other MCP servers capable of manipulating your environment** (e.g., via file system access, HTTP fetches or cloud tools) in the same session. If not controlled, sensitive data could be exposed, stolen or sent to external domains.

It is still advisable to follow general MCP security measures (e.g., **authentication**, **TLS**, **least privilege**). MCP does not provide built-in authentication or encryption, may run over plain HTTP without TLS, and inherits common API security risks. As a result, developers must implement proper access control, data protection, rate limiting and input validation.
Below are some security risks related specifically to **Chrome DevTools MCP server** integration that you should be aware of.

1. Cross-server Data Leak (from DevTools MCP to another MCP server)

Risk: If your MCP Client (e.g., Cursor) connects to **DevTools MCP** and also to another **untrusted or malicious MCP server**, the AI agent could **unintentionally forward** sensitive tab data (tokens, cookies, PII, console/network content) to that other server during tool use.

Safety Actions:

- Enable **only trusted MCP servers** in the MCP Client for that job and disable everything else.
- Enforce a **tool allowlist** for the job. For example, for the specific job **allow** Chrome DevTools MCP tools and **block** file system manipulation MCP tools.
- When choosing an MCP server, consider looking into trusted sources like the MCP official registry under github.com/mcp.

2. Access to Sensitive and Personal Data

Risk:

- Within the active Chrome browser tab, Chrome DevTools MCP can read the DOM, console, network traffic (including request bodies, if allowed) and take screenshots. You must assume the **AI agent sees everything DevTools sees**.
- Moreover, Google collects usage statistics (such as tool invocation success rates, latency and environment information) to improve the reliability and performance of Chrome DevTools MCP. Data collection is enabled by default.

This poses **a significant risk if you are logged into a personal or production account** in the browser tab controlled by the AI agent, as you could **unintentionally expose sensitive information to the AI model**.

Safety Actions:
For the browser session controlled by the AI agent with Chrome DevTools MCP tools:

- Opt out of Google collecting your data if your application is handling sensitive data. You can use the `--no-usage-statistics` flag in your MCP JSON configuration file to do this.
- Do **not** use personal accounts or real production dashboards. Instead, use **sandbox/test accounts** and **masked mock data** only.
- Use a **temporary Chrome profile** for running the tests.
  You can use the `--isolated` flag to achieve this: `chrome-devtools-mcp@latest --isolated`. This flag launches Chrome with a temporary, isolated user profile that is automatically deleted when the session ends, leaving no data behind.
- Ensure the browser tab is closed (automatically or manually) after the session is finished to prevent accidental personal use later.

3. Saved Artifacts (Traces, HAR*, Console Logs) May Contain Secrets

Risk: Traces, HAR files, and console logs that Chrome DevTools can access often include `Authorization` headers, cookies, tokens, emails/IDs or full payloads. If the AI writes these logs and traces to physical files, there is a risk of exposing sensitive data in your version control system or to unintended parties.

Safety Actions:

- **Implement a pre-processing step:** Before any AI processing or uploads, automatically redact sensitive patterns (e.g., `Authorization: Bearer *`, `apiKey=*`, `token=*`, `Set-Cookie`, emails, IDs).

- Consider writing a script to **exclude response bodies** in HAR* files by default. If they are needed, **truncate** and **hash** large bodies.
- Store artifacts (logs, traces, HAR files, reports) **only in a private, temporary storage location** used by your CI/CD system. Do not keep them permanently, instead configure automatic deletion after a short time.
- When sharing results (e.g., in a pull request), include **only summaries or high-level reports**. If someone needs the full files, share a **secure link** that requires authorization to access them.

Chrome DevTools MCP Server: Common Use Cases

As mentioned, since the Chrome DevTools MCP server is in public preview, it is not yet stable enough to be confidently applied to production (at least by the time of publishing this handbook). It is also worth mentioning that AI models sometimes make mistakes and demonstrate inconsistent behavior.
However, the Chrome DevTools MCP server unlocks a lot of potential use cases which teams can start exploring in their local or lower development environments. Below are only a few from all potential use cases that the Chrome DevTools MCP server enables for engineers:

1. Instant End-to-end (E2E) Testing from your IDE and in CI/CD

You can instantly create and run E2E tests without setting up complex frameworks like **Playwright, Cypress or Selenium**. Since the Chrome DevTools MCP server has **Puppeteer** built-in, the AI agent can translate your plain English commands into browser automation instructions.
While this method cannot replace a real framework-based automation setup due to potential instability, it is still perfect for quick validation during development in your IDE or for running smoke tests in a CI/CD pipeline before deployment in lower environments.

- **How it works in your IDE:** You give a natural language prompt directly in your IDE's Agent chat. The AI agent uses the MCP server to control a browser, execute the steps and report back the results.
- **How it works in CI/CD:** When a developer opens a pull request, a service like **GitHub Actions** can automatically run these AI-generated test scripts. The script launches a **headless Chrome browser** on the server, performs the validation steps, and if a test fails (e.g., a button isn't found or a success message doesn't appear), the pipeline fails. This acts as an automated smoke check, helping catch regressions before code is merged.

We explored this workflow in the Cursor Workflow: AI-assisted E2E Test Case with Report Generation section above.

2. Configuring AI Agents for Self-testing

You can configure your coding agent (e.g., Cursor) to use Chrome DevTools MCP server to run the UI in the browser and check the features after implementing them. This allows the AI agent to verify its own changes after implementation and ensure the feature quality.
We explored this workflow in the Cursor Workflow: Configuring AI Agents for Self-testing section above.

3. Autonomous Real-time Staging / Pre-production Monitoring

This is a standalone AI application that acts as a **synthetic user**, continuously checking the health of your application in a **staging / pre-production** environment. Instead of waiting for users to report an issue, this AI agent proactively finds problems in lower environments.

- **How it works:** A scheduled service (e.g., running every 15 minutes) uses the Chrome DevTools MCP server to navigate through critical user flows on your application (like login, searching and checkout). It collects logs, network requests and performance metrics. The data is then fed to an AI model to generate reports and suggest action items.
- **Example AI Alert (posted to Slack):**

```
Critical Alert: The checkout flow is broken. The `/api/payment` endpoint has
returned a `502 Bad Gateway` error for the last 3 consecutive checks.
The page is showing a generic "Something went wrong" message to users. Paging
the on-call engineer.
```

Later, when the Chrome DevTools MCP server is stable and all security concerns are addressed, this use case can also be applied to production environments.

4. Automated Bug Reproduction and Root Cause Analysis

This is a powerful workflow for developers. When a bug is reported, you can ask the AI agent in your IDE to reproduce it. It will perform the steps in the browser using the Chrome DevTools MCP server, gather all the required context (console errors, failed network requests, screenshots), and then use its knowledge of your codebase to pinpoint the exact problem and suggest a fix.

- **Example Prompt:**

```
There's a bug where the user profile page crashes if the user has no avatar.
Please navigate to a user profile without an avatar, capture the console error
and the failed API response and then show me the exact line in
`ProfileComponent.tsx` that is causing the crash.
```

5. Performance Trace Analysis

You can ask the AI agent to run a performance trace, analyze the results and investigate specific performance issues (e.g., high LCP) from your IDE.

- **Example Prompt:**

```
Run a performance profiling audit on http://local.react-note-app.com:3000/notes.
Give me actionable steps to improve its performance.
```

6. On-demand Accessibility, Performance, Best Practices and SEO Audits (Upcoming)

You will be able to ask the AI agent to perform a comprehensive audit directly from your IDE. The **Chrome DevTools MCP** server will support programmatically running **Google's Lighthouse** tool to reveal potential gaps in accessibility, performance (Core Web Vitals), best practices and SEO. With this rich information received from **Chrome DevTools MCP**, your AI agent that has access to your codebase can then translate the generic report into concrete code suggestions tailored to your components.

- **Example Prompt:**

```
Run a Lighthouse accessibility audit on the product details page.
List all images missing `alt` text and suggest descriptive text for them based
on the component's props. Also, check for any color contrast issues.
```

Note: Chrome DevTools MCP cannot run **Google's Lighthouse** tool at the moment of publishing this handbook. However, this feature is included in the project roadmap and should be released soon.
You can follow the official reference to stay updated.
Meanwhile, with access to the browser, your **AI agent** can still give you actionable accessibility suggestions without running the **Lighthouse** tool. If you ask your AI agent to check for accessibility issues (without explicitly requesting a Lighthouse audit), it will perform an accessibility analysis using the page's accessibility tree and custom contrast checking scripts, inspect the DOM for common accessibility issues and give you actionable steps for improvement.

9. Debugging Tools Summary

Tool Name	Common Use Cases
Console Panel	- Get an immediate overview of errors and warnings. - Use for quick debugging and inspecting variable values (e.g., via `console.log`, `console.table`). - Live-test JavaScript code snippets and manipulate the DOM.
Elements Panel	- Live-edit HTML (DOM) and CSS for real-time prototyping. - Manipulate the DOM tree (add/remove nodes and classes). - Inspect element states like `:hover` and `:focus`. - Debug layouts and perform pixel-perfect design checks. - Audit for accessibility issues.
Toggle Device Toolbar	- Test responsive design across different device resolutions. - Simulate touch events. - Perform pixel-perfect testing across various devices to ensure the UI matches the design file (e.g., Figma).
Performance Panel	- Record and analyze runtime performance to find root cause of slow interactions, poor page load and rendering bottlenecks (jank). - Block third-party scripts and compare performance impact. - Annotate a performance trace and attach to a bug ticket or share with the team for further review. - Find root cause of memory leaks. - Identify unused or inefficient code.
Lighthouse Panel	- Run automated audits for performance, accessibility, SEO and web best practices to get actionable reports. - Identify unused JavaScript code.
Network Panel	- Inspect all network requests (headers, payloads) to debug API failures (CORS, Authorization). - Analyze resource load times and verify caching. - Diagnose request latency (e.g., server wait time). - Simulate slow network conditions. - Export network activity as an HAR file and attach to a bug ticket or share with the team for further review.

Tool Name	Common Use Cases
Application Panel	- Inspect and debug cookies and all client-side storage (LocalStorage, SessionStorage, IndexedDB). - Reset the entire application state by clearing site data. - Debug service workers and Progressive Web App (PWA) features along with push messages, notifications and manifest files. - Check if your web application benefits from bfCache (back/forward cache) optimization. - Inspect speculation rules to see which documents were prefetched or prerendered and which ones failed.
Sources Panel	- Review source code file hierarchy to locate the necessary file for debugging or to quickly spot syntax errors. - Edit CSS/JavaScript files in place. - Use Snippets to automate repeated tasks (e.g., performance monitoring tasks).
Local Overrides	- Override response headers (`Cache-Control`, `Access-Control-Allow-Origin`) to do appropriate local testing without waiting for backend fixes. - Mock API responses to test changes locally before pushing to production. - Override source files (HTML, CSS, JS) and check some issues (e.g., performance issues like poor LCP, CLS) quickly locally before applying them to your codebase. - Send your local overrides to team members for collaboration or attach them to the bug ticket for further debugging.
AI Innovations Feature (Experimental)	- Get AI-powered explanations and suggestions for CSS, console issues and code snippets, network requests, source code and performance problems.
Browser Debugger	- The best and quickest approach for debugging UI rendering, DOM events, client-side state and hydration issues right in your live browser environment.
IDE Debugging	- Debug multi-layer applications (Next.js, Angular SSR) by seamlessly tracing the execution flow across both client-side and server-side code.
React DevTools	- Debug React-based performance issues such as finding unnecessary re-renders, expensive long-running calculations or the root cause of slow interactions. - View the overall hierarchy of your components and inspect detailed props and source information. - Debug lazy loaded components and their fallback states (skeleton, loader, or fallback image). - Emulate different scenarios for more convenient debugging (e.g., trigger Suspense fallback, force the selected component into an error state).

Tool Name	Common Use Cases
React Profiler API (Programmatic Profiling)	- Programmatically measure the rendering performance of components to identify bottlenecks and validate optimizations. - Collect real-world performance data from production and send to a performance monitoring platform (e.g., Datadog, Sentry). - Get a deeper understanding of React component lifecycle (mount and update) and interactions.
Angular DevTools	- Record a performance profile and analyze performance bottlenecks (e.g., dropped frames, slow interactions, redundant calculations). - Debug SSR hydration, signal relationships and dependency injection. - Inspect component hierarchy and component details. - Export a Profiler trace and attach to a bug ticket or share with the team for further review.
Redux DevTools	- Debug client-side state for major Flux-style libraries (e.g., Redux, Zustand) by inspecting the history of all actions, viewing state changes, dispatching custom actions and replaying interactions. - Download the recorded timeline of state changes and attach to a bug ticket or share with the team for further review.
PerformanceObserver Interface	- Create browser snippets to monitor the performance of your web application. - Collect essential performance data from your application and send it to real-time monitoring platforms.
Performance: now() Method (High-precision Timing)	- Accurately measure the execution time of JavaScript code blocks. - Simulate long-running tasks to test UI responsiveness.
Front-end Monitoring	- Debug environment-specific issues using captured errors and context from diverse real-user environments. - Investigate non-reproducible bugs using logs and session replays. - Monitor overall application health and get alerted on critical production issues. - Track Core Web Vitals (LCP, CLS, INP) in real-time on real devices using Real User Monitoring (RUM).

Tool Name	Common Use Cases
Chrome DevTools MCP Server (Public Preview)	- Instantly create and run E2E tests directly from your AI coding agent (e.g., Cursor) or in a CI/CD pipeline using plain English prompts without setting up complex frameworks (e.g., Playwright, Cypress). - Configure the AI agent to launch the browser and verify features after implementing them. - Create a standalone AI application that acts as a synthetic user, continuously checking the health of your application in a staging / pre-production environment. - Ask the AI agent to reproduce a bug (e.g., API failure) or do root cause analysis by performing necessary steps in the browser. - Perform accessibility, performance, best practices and SEO audits right from your AI coding agent.

10. Technical Configuration

The following specific personal projects and tools were used to validate debugging techniques in this handbook and to take screenshots. UI elements in DevTools or feature behaviors may vary slightly in other versions.

Proof of Concept (POC) Applications

- **Angular Shop POC** (Custom Signal Store)
  - Angular: `v20.3.15`
  - Node.js: `v22.17.0`
- **Next.js Note Management POC** (Zustand Store)
  - Next.js: `v16.0.3`
  - React: `v19.1.0`
  - Node.js: `v22.17.0`
- **React Todo POC** (Redux Store)
  - React: `v19.1.0`
  - Node.js: `v22.17.0`

> **Note:** References for these projects in my GitHub profile will be provided in later versions of the handbook. Once ready, they will also be linked in the official repository `README` file. You can check the repository for the latest version and updates.

Browser and Extension Versions

- **Google Chrome:** `v143.0.7499.193`
- **React DevTools:** `v7.0.1` (Oct 2025)
- **Redux DevTools:** `v3.2.10`
- **Angular DevTools:** `v1.7.0`

IDE Versions

- **WebStorm IDE:** `2025.2` (Aug 2025)
- **Cursor IDE:** (Dec 2025 - Jan 2026)

11. Documentation and Resources

This handbook **combines my real-world engineering experience with authoritative resources**: official specifications, standard documentation and deep dives by community experts (GDEs and Core Teams). Below you can find the resources I relied on and recommend for further learning.

Official Documentation and Best Practices

Name	Description	Url	Source	License Note
MDN Web Docs	Web APIs (e.g., Service Worker API, Performance APIs, etc.), JavaScript and Web core concepts.	developer.mozilla.org	Mozilla / Community	CC-BY-SA 2.5
Web Development Best Practices	Best practices for building beautiful, accessible, fast and secure websites that work cross-browser.	web.dev	Google Chrome Team and External Experts	CC BY 4.0
Chrome Developers	Chrome DevTools: Official documentation and best practices.	developer.chrome.com	Google Chrome Team	CC BY 4.0
React.dev	React library and React DevTools: Official documentation and best practices.	react.dev	Meta / React Team	CC BY 4.0
Next.js	Next.js framework: Official documentation and best practices.	nextjs.org	Vercel Team	MIT License
Angular.dev	Angular framework and Angular DevTools: Official documentation and best practices.	angular.dev	Google / Angular Team	CC BY 4.0
Redux	Redux and Redux DevTools: Official documentation and best practices.	redux.js.org	Dan Abramov and the Redux Documentation Authors	MIT License

Expert References (GDE and Core Teams)

Name	Description	Url	Source
How modern browsers work	A web developer guide to browser internals (Parsing, Styling, Layout, Painting / Rasterization, Compositing, GPU Rendering).	addyo.substack.com/p/how-modern-browsers-work	**Addy Osmani**
Guide to implementing speculation rules for more complex sites	The costs of speculations, how and when to implement the Speculation Rules API.	developer.chrome.com/docs/web-platform/implementing-speculation-rules	**Barry Pollard**
Web Performance Snippets	A curated collection of JavaScript snippets to measure and debug Web Performance directly in your browser's DevTools Console.	webperf-snippets.nucliweb.net	**Joan León**
Performance Debugging with DevTools: A Practical Deep Dive	A practical session on how to use Chrome DevTools to fix website performance bottlenecks. Covers essential workflows and techniques, including analyzing runtime performance, optimizing code delivery and additional best practices.	youtu.be/8VITEL9tHpQ	**Umar Hansa**
Measure performance with the RAIL model	Explains RAIL (response, animation, idle, and load) model and explores various key performance metrics related to user experience (**goals**), and recommendations that help you achieve goals (**guidelines**).	web.dev/articles/rail	**Chrome Team**
Angular DevTools	A deep-dive talk into Angular DevTools: from its installation, through debugging techniques, identifying rendering bottlenecks, inspecting injectors and more.	youtu.be/BTanGJU9iJc	**Sam Vloeberghs**

Browser Extensions

Name	Chrome	Firefox Add-ons	Microsoft Edge Add-ons
React DevTools	Chrome Web Store	Firefox Add-ons	Microsoft Edge Add-ons
Angular DevTools	Chrome Web Store	Firefox Add-ons	
Redux DevTools	Chrome Web Store	Firefox Add-ons	Microsoft Edge Add-ons

12. About the Author

Hello! I'm Lala Hakobyan, a Senior Front-end Engineer with over 15 years of experience in the tech industry, with the last 9 years in front-end engineering. Throughout my career, I have held different roles: **Full-stack Engineer**, **Project Manager**, **Front-end Engineer** and **Software Engineering Team Lead**. The experience gained from this journey allows me to bridge the gap between engineering and product, ensuring technical decisions align with user needs.

While I enjoyed and fully committed myself to every role, front-end engineering remains my true passion. I'm passionate about delivering seamless experiences to end users and building scalable, maintainable and high-quality applications, from architecture to implementation.

Throughout my career, I've specialized in technologies like **Angular**, **TypeScript** and **Micro-frontend architecture**. Recently, I broadened my stack to include **React** and **Next.js**, leveraging **AI** to enhance engineering productivity. During my career, I continuously shared knowledge with peers, maintained technical documentation and contributed to establishing engineering best practices. This handbook is one of the independent initiatives I took on in 2025: a project that combines **my passion for innovation and knowledge sharing**. It is my way of sharing the practical debugging techniques and insights I've learned from solving complex problems in real-world projects, including modern techniques I adopted during the past year.

Thank you for reading this handbook! I hope you enjoyed it and found it helpful in your everyday debugging flows.
If you found this handbook valuable, consider **starring the GitHub repository** or **sharing the project link** with your teammates to help more people discover it:
github.com/lala-hakobyan/front-end-debugging-handbook

I'd love to hear from you. For feedback, questions, updates or just to connect over meaningful tech conversations, you can find me here:

- linkedin.com/in/lala-hakobyan
- github.com/lala-hakobyan

> **Important Note:** This copy is the FREE version of the handbook for the developer community. If you see this handbook for sale on any platform, please inform me via LinkedIn so that I can prevent its illegal usage.