Using the Web Developer Tools in Firefox

A Complete 11-Lesson Course

Inspector · Console · Network · Debugger · Storage · Performance

Plus four real-world scenario lessons: Authentication, Client Access, Debugging a Slow Page, and Reverse-Engineering an API.

11 Lessons · 7 Panel deep-dives · 4 Real-world scenarios

Contents

LESSON 01 OF 11 · PANEL

Introduction to Firefox DevTools

Overview

Firefox Developer Tools (DevTools) is a set of built-in tools for inspecting, debugging, and profiling web pages — directly inside the browser, with no installation required. Every modern browser has a version of DevTools; Firefox's is one of the most fully featured, with a number of tools — such as the CSS Grid Inspector and Shape Path Editor — that are unique to Firefox.

This lesson introduces the DevTools environment: how to open it, how to navigate between panels, and what each panel is for. The goal is to give you a mental map of the entire toolset before diving into individual panels in lessons 2–7.

Opening DevTools

There are three ways to open DevTools in Firefox:

Method	Result
F12	Opens DevTools to the last-used panel
Ctrl + Shift + I (Windows/Linux) / Cmd + Option + I (macOS)	Opens DevTools to the last-used panel
Right-click any element on the page → Inspect Element	Opens DevTools with the Inspector panel focused on that element

DevTools opens docked to the bottom of the browser by default. You can change the docking position using the `···` menu in the DevTools toolbar: dock to the left, right, bottom, or pop out into a separate window. Many developers prefer docking to the right for wide-screen work — it gives both the page and the DevTools more vertical space.

The DevTools Interface

When DevTools is open, you will see:

The toolbar — a row of panel names along the top. Click any name to switch panels.

The main panel area — the content of whichever panel is currently selected.

The Console drawer — a persistent Console that can be shown at the bottom of *any* panel by pressing `Escape`. This lets you, for example, inspect the DOM in the Inspector while also running JavaScript queries in the Console — without switching panels.

The toolbox options (■) — a settings menu on the far right of the toolbar where you can enable additional panels (such as the Network Monitor if it's hidden), set themes (Light or Dark), and configure advanced settings.

The Panels at a Glance

Firefox DevTools has the following main panels:

Inspector

Inspect and modify the HTML structure (DOM) and CSS styles of any element on the page. Select elements, view and edit applied CSS rules, and see computed values. Covered in **Lesson 2**.

Console

A JavaScript REPL (Read-Eval-Print Loop) that also displays all log messages, errors, warnings, and network errors produced by the page. The single most-used panel in day-to-day development. Covered in **Lesson 3**.

Network

Monitors every HTTP/HTTPS request the page makes: resources, API calls, images, fonts. Shows timing, headers, request and response bodies, and status codes. Covered in **Lesson 4**.

Debugger

A full JavaScript source-level debugger. Set breakpoints, step through code, inspect variable values at any point in execution, and work with source maps. Covered in **Lesson 5**.

Storage

Inspect and edit all client-side storage: Cookies, localStorage, sessionStorage, IndexedDB, and Service Worker Cache. Covered in **Lesson 6**.

Performance

Records a detailed profile of what the browser's main thread was doing — JavaScript execution, layout, paint, composite — during a page load or user interaction. Identifies long tasks and rendering bottlenecks. Covered in **Lesson 7**.

Memory

Records heap snapshots and allocation timelines to find JavaScript memory leaks. Not covered in this course but worth knowing exists.

Accessibility

Audits the page's accessibility tree — what assistive technologies like screen readers see. Not covered in this course.

Network Monitor (same as Network)

Some Firefox versions label this differently — it is the same as the Network panel.

Keyboard Shortcuts Worth Memorising

These shortcuts work regardless of which panel is open:

Shortcut	Action
F12	Toggle DevTools open/closed
Escape	Toggle the Console drawer in the current panel
Ctrl + Shift + M	Toggle Responsive Design Mode (mobile viewport simulation)
Ctrl + P (in Debugger)	Search for a source file by name
Ctrl + F (in Network)	Search inside all response bodies
Ctrl + Shift + R	Hard reload — bypasses cache

Responsive Design Mode

Press `Ctrl + Shift + M` (or click the phone/tablet icon in the DevTools toolbar) to enter Responsive Design Mode. This replaces the normal browser viewport with a resizable frame where you can:

- Set exact pixel dimensions (e.g. 375 × 812 for an iPhone 14)
- Choose from a dropdown of preset device sizes
- Throttle the network to simulate mobile data speeds
- Enable touch event simulation

This mode is essential for testing mobile layouts without needing a physical device.

The Dark and Light Theme

Firefox DevTools defaults to matching the browser's theme. You can force it to Light or Dark via **Settings (■) → Themes**. The examples throughout this course use the **Dark** theme.

DevTools and Page Performance

Opening DevTools has a small impact on page performance — the browser does more bookkeeping to support live inspection. For performance profiling (Lesson 7), this is worth being aware of. In practice it rarely matters for debugging work, but if you are measuring exact millisecond timings you should close and reopen

DevTools to get a clean baseline.

Exercises

1. Open Firefox and press `F12`. What panel opens by default? Look at the toolbar — how many panels are visible?

2. Right-click any element on this page (a heading, a paragraph, a link) → **Inspect Element**. Does DevTools open with the Inspector panel focused on that element?

3. With DevTools open on any panel, press `Escape`. Does the Console drawer appear at the bottom? Type `2 + 2` and press Enter. Does it return `4`?

4. Press `Ctrl + Shift + M` to enter Responsive Design Mode. Use the width dropdown or drag the handles to set the viewport to 375px wide. How does the page layout change?

5. Click the ■ icon in the DevTools toolbar. Can you find the option to switch between Light and Dark themes? Try switching and observe the change.

6. Click the `···` menu next to the ■ icon. What docking options are available? Try docking DevTools to the right side of the browser.

Key Takeaways

- `F12` opens and closes DevTools. **Right-click** → **Inspect Element** opens directly to the Inspector.
- DevTools has seven primary panels: Inspector, Console, Network, Debugger, Storage, Performance, and Memory.
- The **Console drawer** (`Escape`) is available in every panel — use it to run JavaScript queries without leaving the current panel.
- **Responsive Design Mode** (`Ctrl + Shift + M`) simulates mobile viewports without a physical device.
- The `···` **menu** controls docking position; ■ **Settings** controls themes and which panels are visible.

What's Next

Lesson 2 covers the **Inspector panel** in depth — selecting elements, reading and editing the DOM, and understanding the CSS Rules panel and the box model visualiser.

LESSON 02 OF 11 · PANEL

The Inspector Panel

Overview

The Inspector is the panel you will use most often. It shows the live HTML structure of any web page and, alongside it, all the CSS rules that apply to any element you select. You can edit both the HTML and the CSS directly in the panel and see the results instantly on the page. This lesson covers navigating the HTML tree, selecting elements, editing styles, reading the CSS rules pane, using the box model diagram, and the computed styles view.

Opening the Inspector

Three ways:

- Press `Ctrl + Shift + C` to activate the **element picker** — hover over anything on the page and click it. The Inspector opens with that element already selected.
- Right-click any element on the page and choose **Inspect**.
- Press `F12` and click the **Inspector** tab.

The element picker is the fastest route when you know what you want to look at.

The HTML Tree (Left Pane)

The left side of the Inspector shows the page's HTML as a collapsible tree. Each node in the tree is a real HTML element — tag name, attributes, and content.

Navigating the Tree

- **Click** a node to select it. The right pane updates to show its CSS rules.
- **Click the triangle** (■) to expand or collapse a node's children.
- **Alt + Click** the triangle to expand all descendants at once.
- **Arrow keys** navigate up and down through the tree when a node is focused.
- **Right-click** a node for a context menu with options to copy, delete, duplicate, or scroll the element into view.

Searching the Tree

Press `Ctrl + F` while the HTML pane is focused to open a search bar. You can search by:

- **Tag name** — e.g. `nav` or `button`
- **CSS selector** — e.g. `.menu-item` or `#header`
- **Attribute** — e.g. `[data-id="42"]`
- **Text content** — any visible text on the page

Results are highlighted in the tree, and you can step through them with Enter / Shift + Enter.

The Breadcrumb Bar

At the bottom of the HTML pane sits a breadcrumb trail showing the path from the root to the currently selected element:

```
html > body > main > section.hero > h1
```

Click any ancestor in the breadcrumb to jump to it. This is useful when you are deep inside a heavily nested layout and need to move up quickly.

Selecting and Editing HTML

Editing an Attribute

Double-click any attribute value to edit it inline. For example, double-click a `class` attribute and add or remove class names — the page updates immediately.

Editing Text Content

Double-click the text inside a text node to edit it. Useful for checking whether a layout breaks when text is longer or shorter.

Adding a New Attribute

Click a node to select it, then double-click in the space after the last attribute (inside the opening tag). A text cursor appears — type the new attribute and value: `data-debug="true"` and press Enter.

Editing as HTML

Right-click a node → **Edit as HTML**. A text editor opens showing the full HTML of that element and all its children. You can paste in entirely new markup. Press `Ctrl + Enter` (or click outside) to apply.

Moving Elements

Drag any node in the HTML tree to reorder it. You can drag a `<div>` before or after its siblings, or into a different parent. This is handy for testing layout changes without touching source files.

Deleting an Element

Select a node and press `Delete`. The element disappears from the page. Press `Ctrl + Z` to undo.

The CSS Rules Pane (Right Side)

When you select an element in the HTML tree, the right side of the Inspector shows all the CSS rules that apply to it — the element's own styles, styles inherited from parent elements, and the browser's default stylesheet.

Reading the Rules Pane

Rules are listed from **most specific to least specific**. The rule at the top (closest to the element) wins over rules below it.

```
element ← inline styles (highest priority)
.card .title ← class selector from style.css:42
.title ← class selector from style.css:18
h2 ← element selector from style.css:5
Inherited from .card ← properties from parent
user agent stylesheet ← browser defaults
```

Struck-through declarations are overridden — a more specific rule above them is winning. This is the fastest way to answer "why isn't my CSS working?".

Toggling a Declaration

Click the checkbox to the left of any CSS property to toggle it off and on. The page updates instantly. Use this to check what a style is actually doing.

Editing a Value

Click any CSS value to select it. A text cursor appears — type a new value and press Enter. Use the up/down arrow keys to increment/decrement numeric values (e.g. font-size, padding) by 1; hold `Shift` to step by 10.

Adding a New Declaration

Click the empty space at the bottom of a rule block to place the cursor there, then type a new property: value pair.

Adding a New Rule

Click the **+** icon in the Rules pane toolbar to add a new CSS rule targeted at the currently selected element. Firefox generates a selector for you.

Filtering CSS Properties

Type in the **Filter Styles** box at the top of the Rules pane to search for a property by name. Useful on elements with dozens of rules — type `padding` to see only padding-related declarations.

The Box Model Diagram

Switch to the **Layout** tab in the right pane to see the box model diagram. It shows the selected element's actual pixel dimensions in concentric rectangles:

Each layer shows its actual pixel value. Click any value in the diagram to edit it directly. This is faster than hunting for the property in the Rules pane.

What to Look For

- **Unexpected margin collapsing** — two stacked elements sharing one margin instead of having two separate ones.
- **Padding on the wrong side** — a gap that looks like margin but is actually padding, or vice versa.
- **Width not matching expectation** — check whether `box-sizing: border-box` or `content-box` is in effect (shown in the Layout tab).

The Computed Styles Tab

Switch to the **Computed** tab in the right pane to see the **final computed value** of every CSS property on the selected element — after all cascading, inheritance, and specificity has been resolved.

This is different from the Rules pane, which shows the raw declared values. Computed shows what the browser actually uses.

Using Computed Styles

- **Search by property** — type in the filter box to narrow down the list.
- **Show only changed values** — tick the "Browser styles" checkbox to hide browser defaults and see only properties that have been explicitly set.
- **Click the expand arrow** (■) next to a property to see which CSS rule produced that computed value and which file it came from.

When to Use Computed vs Rules

Situation	Use
"Which rule is winning?"	Rules pane — shows the cascade

Situation	Use
"What is the actual pixel value?"	Computed — shows the resolved value
"Why is my font-size 14px when I set 1rem?"	Computed — shows the resolved rem value
"What colour does the browser actually render?"	Computed — shows the final hex/rgb

---

The Changes Pane

Any edits you make in the Inspector are tracked in the **Changes** pane (accessible from the three-dot menu or via the Changes tab if enabled). It shows a diff of every CSS modification you've made in the current session — property and value, old vs new — so you can copy the changes to your actual source files when you're happy with the result.

Practical Tips

Finding which file a style comes from

Every CSS declaration in the Rules pane has a file name and line number to its right (e.g. `style.css:42`). Click it to jump straight to that line in the Debugger's source view.

Forcing element states

Click the **:hov** button in the Rules pane toolbar to force pseudo-states on the selected element: `:hover`, `:focus`, `:active`, `:visited`. This lets you inspect styles that only appear when the user interacts — without having to keep the mouse in exactly the right position.

Screenshot of a single element

Right-click any node in the HTML tree → **Screenshot Node**. Firefox saves a PNG of exactly that element — even if it's partially off-screen.

Copying a CSS selector

Right-click a node → **Copy** → **CSS Selector**. Firefox generates a unique selector for that element which you can paste into your JavaScript or test suite.

Exercises

1. Open DevTools on any website using `Ctrl + Shift + C`, click a heading or navigation link. In the Rules pane, find a colour or font-size property and change its value. Observe the live update. Press `Ctrl + Z` to undo.

2. In the Rules pane, find any property and click its checkbox to toggle it off. Note how the element changes. Toggle it back on.

3. Click the **:hov** button and force `:hover` on a navigation link or button. Look at the Rules pane to see which styles only appear on hover.

4. Select any block element (a `<div>` or `<section>`), switch to the **Layout** tab, and read its box model. Identify the content size, padding, border, and margin values.

5. Switch to the **Computed** tab with the same element selected. Search for `font-size`. Click the expand arrow to see which rule is producing that computed font size.

6. Right-click any element in the HTML tree → **Edit as HTML**. Change a heading's text to something else. Press `Ctrl + Enter` to apply. Refresh the page to confirm the change is not saved.

Key Takeaways

- `Ctrl + Shift + C` activates the element picker — the fastest way to inspect any visible element.
- **The Rules pane** shows all CSS rules from most to least specific; struck-through declarations are overridden.
- **Toggle checkboxes** next to CSS properties to check what a style is actually doing.
- **The box model diagram** (Layout tab) shows exact pixel values for content, padding, border, and margin.
- **Computed styles** show the final resolved value after all cascading — use it when a value seems wrong.
- **:hov** forces pseudo-states so you can inspect hover, focus, and active styles without holding the mouse.
- **File links** in the Rules pane jump straight to the source file and line number.

What's Next

Lesson 3 covers the **Console panel** — reading errors and warnings, using `console.log` and other console methods, running JavaScript directly in the page, and filtering output.

LESSON 03 OF 11 · PANEL

The Console Panel

Overview

The Console is the most immediately useful panel for diagnosing problems on any web page. It shows every JavaScript error and warning the page produces, every message the page's own code logs, and any network failures. It is also a live JavaScript REPL — you can type code directly into the page's context and run it instantly. This lesson covers reading and understanding console output, the console API, filtering messages, running JavaScript interactively, and a handful of techniques that go beyond basic logging.

Opening the Console

- **Keyboard shortcut**: `Ctrl + Shift + K` (macOS: `Cmd + Option + K`) — opens DevTools directly on the Console panel.
- **From any other panel**: press `Escape` to open the Console drawer at the bottom without leaving your current panel.
- **Via F12**: open DevTools, then click the **Console** tab.

The Console drawer (opened with Escape) is particularly useful — you can watch the Network panel and run JavaScript at the same time.

The Console Toolbar

At the top of the Console panel sits a toolbar with several controls:

Control	Purpose
■ (clear)	Clear all current messages. Keyboard: Ctrl + L
Filter box	Type text to show only matching messages
Error / Warn / Log / Info / Debug	Toggle buttons to show/hide each message level
CSS / XHR / Requests	Filter to show only CSS warnings, network requests, etc.
Persist logs	Keep messages across page navigations (not cleared on reload)
Show timestamps	Prefix each message with the time it was produced
Group similar	Collapse repeated identical messages into one with a count

Message Types

The Console uses colour and icons to distinguish message types at a glance:

Errors (■ red)

JavaScript exceptions that stopped execution, failed network requests, and security violations. These always need attention — they mean something genuinely broke.

```
Uncaught TypeError: Cannot read properties of undefined (reading 'name')
at app.js:42:18
```

Click the file and line reference (`app.js:42`) on the right to jump to the Debugger at that line.

Warnings (■ yellow)

Non-fatal issues: deprecated APIs, slow operations, mixed content, missing resources. The page still works, but something should be fixed.

```
Warning: Each child in a list should have a unique "key" prop.
```

Logs (■ grey/white)

Output from `console.log()` calls in the page's own JavaScript. Developers use these intentionally for debugging. On a production site you shouldn't see many of these — if you do, the developer left their debug logging in.

Info (■ blue)

Output from `console.info()`. Similar to log but with an info icon. Browser extension messages and some framework messages appear here.

Debug (grey, hidden by default)

Output from `console.debug()`. Hidden unless you tick the **Debug** filter button. Used for very verbose logging.

Reading an Error Message

A full error entry has several parts:

```
Uncaught TypeError: Cannot read properties of undefined (reading 'username')
at getUserName (app.js:42:18)
at init (app.js:15:5)
at app.js:3:1
```

- **Error type**: `TypeError` — the category of the exception.
- **Message**: `Cannot read properties of undefined (reading 'username')` — what went wrong.

- **Stack trace**: each line shows the function name, file, line number, and column. Read from top to bottom — the top entry is where the error occurred, the entries below show what called it.
- **File link**: click any `app.js:42` reference to jump to that exact line in the Debugger.

Common Error Types

Error	Typical Cause
TypeError	Calling a method on null or undefined; wrong argument type
ReferenceError	Using a variable that hasn't been declared
SyntaxError	Invalid JavaScript syntax (usually in eval or dynamic code)
NetworkError	A fetch/XHR request failed — check the Network panel
404 (Not Found)	A resource (image, script, stylesheet) wasn't found
CORS error	A cross-origin request was blocked — covered in Lesson 9

The Console API

The page's JavaScript can call these methods to write messages to the Console. You can also call them yourself in the REPL:

Basic Logging

```
console.log('User loaded:', user); // general output
console.info('Server responded with 200'); // informational
console.warn('Token expires soon'); // yellow warning
console.error('Failed to fetch user'); // red error (does not throw)
console.debug('Raw response:', rawData); // verbose, hidden by default
```

console.table()

Displays an array of objects as a formatted table — far easier to read than a plain log:

```
const users = [
{ id: 1, name: 'Alice', role: 'admin' },
{ id: 2, name: 'Bob', role: 'user' },
];
console.table(users);
// renders as:
// ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
// ■ (index) ■ id ■ name ■ role ■
// ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
// ■ 0 ■ 1 ■ Alice ■ admin ■
// ■ 1 ■ 2 ■ Bob ■ user ■
// ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
```

console.group() and console.groupCollapsed()

Group related messages together under a collapsible label:

```
console.group('Auth flow');
console.log('Checking token...');
console.log('Token valid, user id:', userId);
console.groupEnd();
```

`groupCollapsed()` starts the group collapsed — useful for verbose output you only expand when needed.

console.time() and console.timeEnd()

Measure how long a block of code takes to run:

```
console.time('data fetch');
const data = await fetchUsers();
console.timeEnd('data fetch');
// → data fetch: 143ms
```

console.count() and console.countReset()

Count how many times a label has been logged:

```
console.count('button clicked'); // → button clicked: 1
console.count('button clicked'); // → button clicked: 2
console.countReset('button clicked');
console.count('button clicked'); // → button clicked: 1
```

console.assert()

Logs an error only if the condition is false — a lightweight assertion for debugging:

```
console.assert(user !== null, 'Expected user to be defined', user);
// logs nothing if user is not null
// logs an error if user IS null
```

console.clear()

Clears all messages programmatically. Equivalent to clicking the clear button or pressing `Ctrl + L`.

Filtering Output

On a busy page the Console can fill with hundreds of messages. Filtering keeps it manageable.

Level Filters

Click the toggle buttons in the toolbar to show or hide each level:

- **Errors** — always leave this on.
- **Warnings** — useful to have on; reveals deprecations.
- **Logs** — can be very noisy on some sites; turn off to focus on errors.
- **Info / Debug** — off by default; turn on for framework verbose output.

Text Filter

Type in the filter box to show only messages containing that text. This supports:

- **Plain text**: `fetch` — shows any message containing "fetch".
- **Regex**: `/auth|login/` — shows messages matching the pattern.
- **Negative filter**: `-analytics` — hides messages containing "analytics". Prefix with `-` to exclude.

Source Filter

Click the **Console Settings** (gear icon) → choose **Show Content Messages Only** to hide browser-generated messages and see only messages from the page's own JavaScript.

Persist Logs

Tick **Persist Logs** in the toolbar to prevent the Console from clearing when the page navigates or reloads. Useful when debugging a flow that involves a redirect — without persist logs, you lose all the messages from the previous page the moment the redirect happens.

Running JavaScript in the Console

The Console input at the bottom is a full JavaScript REPL running in the page's context. Everything on the page is accessible.

Accessing Page Elements

```
// Select elements exactly as you would in code
document.querySelector('h1').textContent
// → "Welcome to My Site"

document.querySelectorAll('.card').length
// → 12

// Read and modify the DOM
document.querySelector('h1').style.color = 'red';
```

Inspecting Variables

If the page's JavaScript defines global variables or exposes them on `window`, you can read them directly:

```
window.userData
// → { id: 1, name: "Alice", role: "admin" }
```

Running Async Code

The Console supports top-level `await`:

```
const res = await fetch('/api/users');
const data = await res.json();
console.table(data);
```

The $() Shortcut

The Console provides `$()` as a shorthand for `document.querySelector()` and `$$()` for `document.querySelectorAll()`:

```
$('h1').textContent // first h1
$$('a').map(a => a.href) // all link hrefs as an array
```

$0 — The Last Inspected Element

`$0` always refers to the element most recently selected in the Inspector. This is extremely useful — select an element in the Inspector, switch to the Console, and interact with it programmatically:

```
$0.classList.toggle('active');
$0.getBoundingClientRect();
// → DOMRect { x: 20, y: 84, width: 320, height: 48, ... }
```

copy()

The `copy()` function copies any value to the clipboard as a string:

```
copy(document.cookie); // copies all cookies
copy(JSON.stringify(window.userData, null, 2)); // copies object as formatted JSON
```

Multi-line Input

For longer code snippets, press `Shift + Enter` to insert a new line without running the code. Press `Enter` alone to execute. Alternatively, click the **split console** icon (if available in your Firefox version) to expand the input area.

Console History

Press the **up arrow** key in the Console input to cycle through previously entered commands — the same as a terminal history. Press **down arrow** to move forward through history.

Practical Tips

Reading a CORS error

A blocked cross-origin request appears in the Console as a red error followed by the request URL. The exact wording tells you whether the server sent no CORS headers, the wrong origin, or blocked credentials. This is covered fully in Lesson 9.

Finding console.log left in production code

Open the Console on any site and filter by **Log** level. If you see a stream of log messages the site is running in development mode or the developer forgot to strip debug logging before deploying.

Storing a value as a global variable

Right-click any logged object in the Console → **Store as Global Variable**. Firefox assigns it to `temp1`, `temp2`, etc. You can then call methods on it, pass it to other functions, or copy it — without re-fetching or re-running the code that produced it.

Exercises

1. Open the Console on any website (`Ctrl + Shift + K`). Look at the existing messages. Are there any red errors? Yellow warnings? Note what they are.

2. Type `document.title` in the Console input and press Enter. Then type `document.querySelectorAll('a').length` and press Enter to count how many links the page has.

3. Run `console.table(performance.getEntriesByType('navigation'))` and read the result. You'll see timing data for the page load as a formatted table.

4. Select an element in the Inspector, then switch to the Console and type `$0.getBoundingClientRect()`. Observe the position and dimensions of the element you selected.

5. Type a multi-line snippet using `Shift + Enter`:

```
const links = $$('a');
links.forEach(a => console.log(a.href));
```

Press Enter to run it. All link URLs on the page appear in the Console.

6. Enable **Persist Logs** and navigate from one page to another within the same site (click an internal link). Confirm that messages from the previous page are still visible in the Console.

Key Takeaways

- **Red errors** mean something broke; **yellow warnings** are non-fatal but should be addressed.
- **Click any file:line reference** in an error to jump straight to the Debugger at that line.
- `console.table()` is the fastest way to read an array of objects — far clearer than `console.log()`.

- `$0` refers to the last element selected in the Inspector — use it to interact with elements programmatically.
- `$$('selector')` is shorthand for `document.querySelectorAll()` and returns a real array.
- **Persist Logs** keeps messages across page navigations — essential when debugging redirect flows.
- **Text filtering with** `-term` hides noisy messages so you can focus on what matters.

What's Next

Lesson 4 covers the **Network panel** — the request list, reading headers and response bodies, understanding status codes, the timing waterfall, and filtering traffic.

LESSON 04 OF 11 · PANEL

The Network Panel & HAR Files

Overview

The Network panel records every HTTP request the browser makes while loading a page — HTML documents, stylesheets, scripts, images, fonts, API calls, WebSocket connections — and shows you the full detail of each one: the URL, method, status code, request and response headers, response body, and a timing breakdown. It is the essential tool for diagnosing slow loads, broken API calls, authentication failures, CORS errors, and unexpected redirects.

Opening the Network Panel

- Press `Ctrl + Shift + E` (macOS: `Cmd + Option + E`) to open DevTools directly on the Network panel.
- Or press `F12` and click the **Network** tab.

Important: the Network panel only records requests made while it is open. Reload the page after opening it to capture the full page load. If you open it after the page has loaded, you will only see requests made from that point onwards.

The Request List

The main area of the Network panel is a table. Each row is one network request. The columns are:

Column	What it shows
Status	HTTP status code (200, 404, 500, etc.)
Method	HTTP verb: GET, POST, PUT, DELETE, etc.
Domain	The hostname the request was sent to
File	The path component of the URL
Initiator	What triggered the request — the HTML parser, a script, CSS, etc.
Type	The resource type: html, json, script, stylesheet, img, font, xhr, fetch, ws
Transferred	Bytes received over the network (compressed)
Size	Actual decompressed size of the resource

Column	What it shows
Time	Total request duration
Waterfall	A proportional bar showing when this request started and how long each phase took

Right-click any column header to add or remove columns. The **Waterfall** column is one of the most useful — it shows at a glance which requests ran in parallel and which were delayed.

Status Codes

The Status column is colour-coded:

Colour	Range	Meaning
Green	2xx	Success — the request completed normally
Blue	3xx	Redirect — the browser was sent to a different URL
Orange	4xx	Client error — the resource wasn't found or access was denied
Red	5xx	Server error — the server failed to fulfil the request
Grey	—	Blocked or cancelled by the browser

Common codes to know

Code	Name	What it means in DevTools
200	OK	Normal success
204	No Content	Success with no response body (common for DELETE)
301 / 302	Moved / Found	Permanent or temporary redirect
304	Not Modified	Browser used its cached copy — no data transferred
400	Bad Request	The server rejected the request (malformed body, missing field)
401	Unauthorized	No valid credentials supplied
403	Forbidden	Credentials supplied but access denied
404	Not Found	The URL doesn't exist on the server
429	Too Many Requests	Rate-limited

Code	Name	What it means in DevTools
500	Internal Server Error	The server crashed or threw an exception
502 / 503	Bad Gateway / Unavailable	Proxy or upstream server problem

A **304** with `0 B` transferred is normal and desirable — it means the browser's cache is working. A string of **401** or **403** responses to API calls usually points to an authentication problem.

Inspecting a Request

Click any row in the request list to open the request detail panel. It has several tabs:

Headers

Shows the full request and response headers.

Request headers — what the browser sent:

- `Authorization` — Bearer token, Basic credentials, or API key
- `Content-Type` — format of the request body (`application/json`, `application/x-www-form-urlencoded`)
- `Accept` — what formats the browser will accept back
- `Cookie` — cookies sent with the request
- `Origin` / `Referer` — where the request came from (important for CORS)

Response headers — what the server sent back:

- `Content-Type` — format of the response body (`application/json; charset=utf-8`)
- `Cache-Control` / `ETag` — caching instructions
- `Access-Control-Allow-Origin` — CORS permission header
- `Set-Cookie` — cookies the server is setting
- `Location` — redirect destination (used with 3xx responses)
- `WWW-Authenticate` — authentication scheme required (used with 401 responses)

Response

Shows the raw response body. For JSON APIs this is the actual data the server returned, formatted with syntax highlighting. If the response is HTML, you see the markup. If it's an image, you see a preview.

This tab is where you check:

- What the API actually returned (versus what your code expected)
- Error messages from the server (often buried inside a JSON body on a 400 or 500 response)
- Whether a redirect landed at the right URL

Request

Shows the request body for POST/PUT requests. Firefox displays it in a parsed form for common content types:

- **JSON** — shown as a formatted, collapsible tree
- **Form data** — shown as key-value pairs
- **Raw** — toggle to see the raw bytes

Cookies

Shows the cookies sent with the request and any `Set-Cookie` headers in the response. Useful for checking whether a session cookie is present, expired, or missing the `HttpOnly` / `Secure` flags.

Timings

A breakdown of time spent in each phase of the request (described in detail in the Timing Waterfall section below).

The Timing Waterfall

The **Waterfall** column in the request list shows a proportional bar for each request. The bar is divided into colour-coded segments — each segment is a phase of the HTTP lifecycle:

```
DNS Lookup ■■
Connecting ■■■■
TLS Setup ■■■■■■
Sending ■
Waiting ■■■■■■■■■■■■■■■■■■■■ ← most of the time is here (TTFB)
Receiving ■■■
```

Phase	What's happening
DNS Lookup	Resolving the hostname to an IP address
Connecting	TCP handshake — establishing the network connection
TLS Setup	HTTPS negotiation — only on secure connections
Sending	Time to upload the request (headers + body)
Waiting	Time between request sent and first byte received — TTFB (Time to First Byte). This is the server's processing time. A long TTFB means a slow server.
Receiving	Time to download the response body

Reading the Waterfall

- **Wide Waiting bars** → the server is slow to respond. Check your backend.
- **Wide Receiving bars** → the response is large. Consider compression or reducing payload size.

- **DNS / Connecting bars appearing on every request** → connection reuse isn't working (missing `keep-alive` or HTTP/2).
- **Requests starting late** → they are blocked waiting for a resource to finish loading first (parser-blocking scripts, for example).
- **A long thin bar at the very top** → the first request is the HTML document. Everything below it started after the HTML was parsed.

DOMContentLoaded and Load

The Network panel overlays two vertical lines on the waterfall:

- **Blue line** — `DOMContentLoaded`: the HTML was parsed and the DOM is ready. JavaScript can run.
- **Red line** — `Load`: all resources (images, stylesheets, scripts) have finished loading.

Everything to the left of the blue line is what blocks the page from being interactive.

Filtering Requests

The toolbar above the request list has several filtering controls:

By type

Click the type buttons to show only a category:

- **All** — every request
- **HTML** — document requests
- **CSS** — stylesheets
- **JS** — JavaScript files
- **XHR/Fetch** — API calls made by JavaScript (the most useful filter when debugging APIs)
- **Fonts** — web font files
- **Images** — image resources
- **Media** — audio and video
- **WS** — WebSocket connections
- **Other** — anything not categorised above

By text

Type in the **Search** box to filter by URL. Regex is supported — e.g. `/api\/users/` to show only requests to the users API endpoint.

By status

Right-click any request → **Filter by Status** to show only requests with the same status code.

Blocking a request

Right-click any request → **Block URL** to tell the browser to refuse that request on the next load. This simulates what happens if a third-party resource (an analytics script, an ad server, a CDN) is unavailable —

useful for checking whether your page degrades gracefully.

Throttling the Network

The toolbar includes a **throttling dropdown** (it may say "No throttling" by default). Use it to simulate slower connections:

Preset	Simulates
GPRS	Very slow mobile (50 Kbps)
Regular 3G	Typical 3G mobile (~750 Kbps)
Regular 4G	Typical LTE (~4 Mbps)
Wi-Fi	Fast connection (~30 Mbps)

Throttling applies to all requests, including API calls. Use it to test whether your loading states, skeleton screens, and error handling actually appear for users on slow connections — they often never appear during development on a fast local machine.

Searching Inside Responses

Press `Ctrl + F` in the Network panel to open the network search box. This searches inside the headers and response bodies of every recorded request — not just the URLs. It's invaluable when you know what data you're looking for but don't know which request returned it. For example, searching for a user's email address will highlight every request whose response body contained it.

HAR Files

What Is a HAR File?

A **HAR file** (HTTP Archive) is a JSON-formatted archive of all the network requests captured in the Network panel. HAR stands for **HTTP Archive** and the file extension is `.har`. It contains a complete record of every request — URL, method, status, all headers, response body, timing data, cookies, and more — serialised to text so it can be saved, shared, and analysed offline.

HAR files are the standard format for sharing network traces between developers, passing them to support teams, or submitting them to performance analysis tools.

What a HAR File Contains

At the top level a HAR file is a JSON object with a single `log` key:

```
{
"log": {
"version": "1.2",
"creator": { "name": "Firefox", "version": "120.0" },
"pages": [ ... ],
"entries": [ ... ]
}
}
```

- `pages` — one entry per page load, with timing for `DOMContentLoaded` and the `load` event.
- `entries` — one object per network request, containing:
- `startedDateTime` — ISO 8601 timestamp of when the request started
- `time` — total duration in milliseconds
- `request` — method, URL, HTTP version, headers, cookies, body
- `response` — status, status text, headers, cookies, body, size
- `timings` — breakdown into `dns`, `connect`, `ssl`, `send`, `wait`, `receive` phases (in milliseconds)
- `serverIPAddress` — the IP address the request was sent to
- `cache` — cache state before and after the request

Creating a HAR File in Firefox

1. Open DevTools and go to the **Network** panel.
2. Reproduce the network activity you want to capture (reload the page, complete the user flow, trigger the API call).
3. Right-click anywhere in the request list → **Save All as HAR**.
- Alternatively, click the **Save** icon (floppy disk) in the toolbar.
4. Save the `.har` file to disk.

To capture a problem that happens during page load, make sure to reload the page *after* opening the Network panel so the panel captures the full load sequence.

Reading a HAR File

A HAR file is plain JSON — you can open it in any text editor. However, it is more practical to load it back into a browser or an analysis tool:

In Firefox:

1. Open the Network panel.
2. Drag and drop the `.har` file onto the request list, or use the import button.
3. The panel replays the recorded requests exactly as they were captured — you can click any request and inspect all its headers, response bodies, and timings as if you had just made the request.

In dedicated tools:

- **HAR Analyzer** (https://toolbox.googleapps.com/apps/har_analyzer/) — Google's web-based tool. Paste in a HAR and it shows a colour-coded waterfall, flags slow requests, and highlights errors.
- **Charles Proxy** — desktop app that can import and display HAR files with a rich GUI.
- **Fiddler** — Windows proxy/debugger that imports HAR files.

- **Webpagetest.org** — can export and import HAR for performance analysis.

Reading a HAR file manually (useful for scripting or automation):

```
import json

with open('trace.har') as f:
har = json.load(f)

for entry in har['log']['entries']:
req = entry['request']
resp = entry['response']
time = entry['time']
print(f"{resp['status']} {req['method']} {req['url']} ({time:.0f}ms)")
```

Using a HAR File

Sharing with a colleague or support team

When you can reproduce a bug but a colleague cannot, capture a HAR file and send it. They can import it into their own browser's Network panel and inspect the exact requests and responses you saw — without needing access to your environment or credentials (though see the security warning below).

Performance analysis

Load the HAR into HAR Analyzer or Webpagetest to get a visual waterfall, identify the largest requests, find requests that were not compressed, and see which resources blocked the critical path.

Regression testing

Store HAR files as fixtures in a test suite. Tools like `jest-fetch-mock` and `msw` (Mock Service Worker) can replay HAR entries to simulate API responses without a real network — useful for testing loading states and error handling.

Comparing before and after

Capture a HAR before and after a performance optimisation. Load both into HAR Analyzer side by side to compare total load time, number of requests, and bytes transferred.

Debugging intermittent issues

Some bugs only happen occasionally or only in production. Ask a user to capture a HAR file during the failure and send it to you. The HAR will contain the exact server responses — including error messages — that the user's browser received.

HAR File Security Warning

HAR files capture **everything** — including:

- **Session cookies** in request headers
- **Authorization tokens** (Bearer, Basic, API keys)
- **Response bodies** which may contain personal data, API keys returned in responses, or confidential business data

Before sharing a HAR file outside your organisation:

- Open it in a text editor and search for `Authorization`, `Cookie`, `Set-Cookie`, and `token`.
- Redact any sensitive values before sharing.
- Firefox's "Save All as HAR" dialog does not automatically redact sensitive headers.

Some tools (Chrome's DevTools, Fiddler) offer a "sanitise" option that strips credentials before export — Firefox does not currently have this built in.

Practical Tips

Spotting unnecessary requests

In the request list, sort by **Size** descending. The largest resources are candidates for lazy-loading, compression, or caching.

Checking compression

Compare the **Transferred** and **Size** columns. If they are the same, the response was not compressed. The server should send `Content-Encoding: gzip` or `br` for text resources (HTML, JSON, JS, CSS). A 400 KB JSON payload with no compression is a missed optimisation.

Confirming caching is working

Filter by type `XHR/Fetch` and look for 304 responses. A 304 means the browser had a cached copy and the server confirmed it was still fresh. If you see 200 responses every time for the same static file, the cache headers are misconfigured.

Finding the real error on a 500

Click a red 500 request → open the **Response** tab. Most servers include an error message or stack trace in the response body. This tells you far more than the status code alone.

Checking authentication headers

Click an API request → **Headers** tab → look for `Authorization` in the request headers. If it's missing on a 401 response, the client isn't sending credentials. If it's present but the response is still 401, the token is invalid or expired.

Exercises

1. Open the Network panel on any website (`Ctrl + Shift + E`), reload the page, and note how many requests were made. Filter by **JS** and then by **XHR/Fetch**. How many JavaScript files and API calls does the page make?

2. Click any request and open the **Headers** tab. Find the `Content-Type` response header. What format is the server returning?

3. Click the **Timings** tab for a request to an external API or image. Which phase (DNS, Connecting, Waiting, Receiving) takes the longest?

4. In the toolbar, set throttling to **Regular 3G** and reload the page. Compare the load time to no throttling. Do loading states or skeleton screens appear?

5. Press `Ctrl + F` in the Network panel and search for a word you know is on the page (the site's name, or "logo"). Which request returned it?

6. Capture a HAR file: open the Network panel, reload a page, then right-click the request list → **Save All as HAR**. Open the file in a text editor and find the first entry in the `entries` array. Read the `timings` object — what was the `wait` time (TTFB) for the HTML document?

7. Load the HAR file you just created back into the Network panel by dragging it onto the request list. Verify that the captured requests are replayed correctly.

Key Takeaways

- The Network panel records every request — **reload after opening** to capture the full page load.
- **XHR/Fetch filter** shows only API calls — the most useful filter when debugging backend communication.
- **Status codes**: 2xx = success, 3xx = redirect, 4xx = client error, 5xx = server error; 304 = cached (good).
- **The Response tab** shows the actual server response body — where error messages are buried on 4xx/5xx.
- **TTFB (Waiting phase)** is the server's processing time — a wide Waiting bar means a slow backend.
- **HAR files** are JSON archives of the entire network trace — save them to share, replay, or analyse offline.
- HAR files contain **session tokens and cookies** — redact sensitive values before sharing externally.
- **Network throttling** simulates slow connections — always test on a throttled connection before shipping.

What's Next

Lesson 5 covers the **Debugger panel** — setting breakpoints, stepping through JavaScript line by line, watching variable values, and diagnosing logic errors without adding `console.log` everywhere.

LESSON 05 OF 11 · PANEL

The Debugger Panel

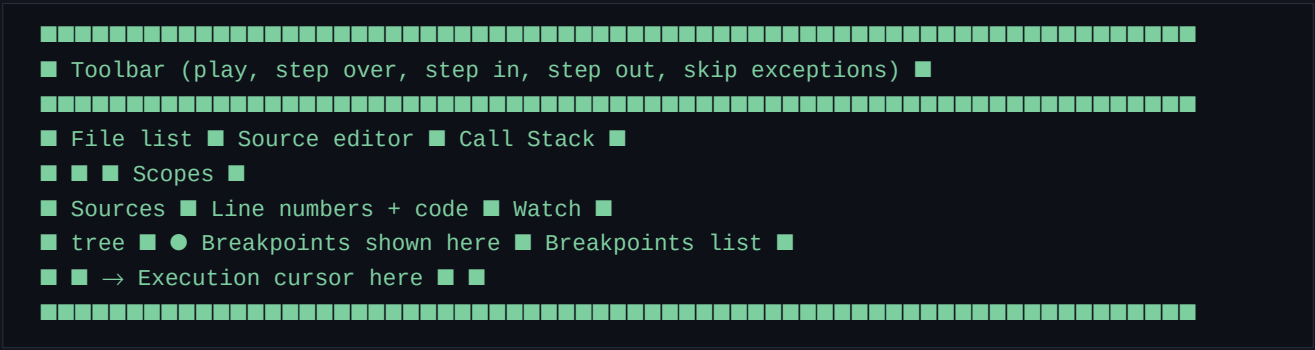
Overview

The Debugger lets you pause JavaScript execution at any point, step through code line by line, inspect the value of every variable, and trace the exact path the program took to reach a problem. It is the tool that replaces endless `console.log()` calls — instead of guessing what values look like at a given moment, you pause the program and read them directly.

Press `Ctrl + Shift + Z` (macOS: `Cmd + Opt + Z`) to open the Debugger directly, or press `F12` and click the **Debugger** tab.

The Debugger Layout

The Debugger is divided into three panes:



- **Left pane** — the file tree showing all JavaScript sources loaded by the page.
- **Centre pane** — the source editor. This is where you read code, set breakpoints, and see the execution cursor when paused.
- **Right pane** — the Call Stack, Scopes, Watch Expressions, and Breakpoints list; these update live while paused.

Opening a Source File

From the file tree

Expand the file tree in the left pane and click any `.js` file. Bundled applications often have a single `bundle.js` or `main.js` — expand it using source maps (described later) to see the original individual files.

With the file search

Press `Ctrl + P` inside the Debugger to open a file search box. Type the name (or part of the name) of the file you want. This is faster than browsing the tree on a large application.

From a Console error

When you see an error in the Console, click the `file.js:42` link at the right of the error message. The Debugger opens with that file and that line highlighted.

From the Network panel

Click a JS request in the Network panel → open the Response tab → click the source link. Or right-click a `.js` file in the Network panel → **Open in Debugger**.

Breakpoints

A breakpoint is an instruction to pause execution when a specific line of code is about to run. When the browser hits the breakpoint, the page freezes, JavaScript stops, and you can inspect every variable in scope.

Line Breakpoints

Click the **line number** in the source editor gutter to add a line breakpoint. A blue dot (●) appears on the line. The next time that line is executed, the browser pauses.

To remove a breakpoint, click the blue dot again.

Conditional Breakpoints

Right-click a line number → **Add Conditional Breakpoint**. Type a JavaScript expression. The breakpoint only pauses when the expression is **true**. This is essential when a bug only occurs for specific data:

```
// Only pause when the user id is 42
user.id === 42

// Only pause when the array is unexpectedly empty
items.length === 0

// Only pause on the tenth iteration
i === 9
```

Conditional breakpoints appear as orange dots instead of blue.

Logpoints

Right-click a line number → **Add Log**. Type an expression or message. Instead of pausing, Firefox logs the result to the Console every time that line runs — like `console.log()` but without touching the source code:

```
"User loaded: " + user.name
```

Logpoints appear as purple dots. They are especially useful when you want to trace a loop or event handler that runs hundreds of times but don't want to pause on every iteration.

Exception Breakpoints

In the toolbar, click the **Pause on Exceptions** button (the icon looks like a hexagon with a pause symbol). When enabled, the Debugger pauses automatically the moment any exception is thrown — before it propagates, before any `catch` block runs, and before any error appears in the Console. This is the fastest way to find where an error originates.

Two sub-options are available:

- **Pause on caught exceptions** — also pauses inside `try/catch` blocks.
- **Pause on uncaught exceptions only** — pauses only when no `catch` block handles the error.

DOM Breakpoints (from the Inspector)

You can set a breakpoint on a DOM mutation from the Inspector panel:

- Right-click any HTML element → **Break On** → choose:
- **Subtree Modification** — pauses when any child element is added, removed, or moved.
- **Attribute Modification** — pauses when an attribute on the element changes.
- **Node Removal** — pauses when the element itself is removed from the DOM.

Firefox jumps to the Debugger and pauses on the exact line of JavaScript that caused the DOM change.

The Stepping Controls

Once paused, a toolbar of controls appears. These let you move through the code at your own pace:

Button	Keyboard	Name	What it does
■	F8	Resume	Continues execution until the next breakpoint or end of script
■	F10	Step Over	Executes the current line and moves to the next. If the current line calls a function, the whole function runs without entering it.
↓	F11	Step Into	If the current line calls a function, moves the cursor *into* that function's body.
↑	Shift + F11	Step Out	Runs to the end of the current function and pauses in the calling function.

Button	Keyboard	Name	What it does
→	—	Step (single instruction)	Advances one statement at a time (useful for complex single-line expressions).

When to use each

- **Step Over** — the default. Move forward without diving into library code you don't care about.
- **Step Into** — use when you want to see what a function you wrote actually does.
- **Step Out** — use when you accidentally stepped into a function you don't care about and want to get back to where you were.
- **Resume** — use when you've seen what you need at one breakpoint and want to run to the next.

The Call Stack

When paused, the **Call Stack** panel on the right shows the chain of function calls that led to the current execution point — the most recent call at the top, the entry point at the bottom:

```
getUserName ← where execution is paused right now
initDashboard ← called getUserName
DOMContentLoaded ← called initDashboard
```

Click any entry in the call stack to jump to that frame in the source editor. The Scopes panel updates to show the variables that existed in that frame at the time of the call. This lets you trace the full path the program took to reach the problem.

The Scopes Panel

While paused, the **Scopes** panel shows every variable accessible from the current execution point, organised into scopes:

- **Local** — variables declared in the current function (including function parameters)
- **Block** — variables declared in the current block (`let`, `const`)
- **Closure** — variables from outer functions that this function has closed over
- **Module** — variables at module scope
- **Global** — properties on the `window` object

Each variable shows its current value. Objects and arrays can be expanded to inspect their properties. You can also **edit** a value directly in the Scopes panel — click a value, type a new one, and press Enter. Execution continues with the modified value.

Watch Expressions

The **Watch** panel lets you type any JavaScript expression and have it re-evaluated automatically every time execution pauses:

```
user.profile.email
items.filter(i => !i.active).length
Date.now() - startTime
```

Watch expressions update every time you step. If an expression throws an error (because a variable isn't in scope), it shows the error rather than a value.

This is more flexible than the Scopes panel because you can compute derived values — not just read raw variables.

Blackboxing (Ignore List)

When stepping through code, you will often accidentally step into framework or library code (React internals, Lodash utilities, bundler boilerplate) that you didn't write and don't care about. **Blackboxing** tells the Debugger to skip over these files.

To blackbox a file

Right-click any file in the file tree → **Ignore Source**. The file turns grey in the tree and the Debugger will never pause inside it, even if a breakpoint would otherwise trigger there.

Blackboxing from a stack frame

While paused, right-click any frame in the Call Stack that belongs to a library → **Ignore Source**. The Debugger skips all future execution in that file.

Built-in framework detection

Firefox DevTools automatically recognises common frameworks and offers to blackbox their source files. You can configure this in Settings → **Ignore List**.

Source Maps

Modern JavaScript is almost always **bundled, minified, and transpiled** before it reaches the browser — a React component written in TypeScript becomes a single unreadable `bundle.min.js`. **Source maps** are companion files (`.js.map`) that tell the Debugger how the minified code maps back to the original source.

When source maps are available

The Debugger automatically loads source maps when the server sends them. Instead of seeing `bundle.min.js`, you see the original file tree — individual components, TypeScript files, SCSS modules — exactly as they were written. You set breakpoints in the original source and the Debugger handles the mapping transparently.

When source maps are not available

You see only the minified output — a single long line of code with mangled variable names. In this case:

- Click the **{ }** (Pretty Print) button at the bottom of the editor to re-format the minified code with line breaks and indentation. It still has mangled names, but it becomes readable enough to follow the logic and set breakpoints.
- Variable names like `a`, `b`, `t` won't match the original source, but you can still inspect their values.

Checking if source maps are loaded

The file tree in the left pane will show individual original files (e.g. `src/components/UserCard.tsx`) rather than just `bundle.js` when source maps are active.

Practical Debugging Workflow

Finding where a value goes wrong

1. Identify the symptom — an incorrect value displayed, a wrong API call made, a UI element in the wrong state.
2. Find the function that should produce the correct value.
3. Set a line breakpoint at the start of that function.
4. Reproduce the issue (click the button, submit the form, reload the page).
5. When paused, read the Scopes panel — is the input data correct? Are all variables what you expect?
6. Step Over each line, watching the Scopes panel update.
7. The moment a variable changes to a wrong value, you've found the bug.

Debugging an event handler

1. Open the source file that registers the event listener.
2. Set a line breakpoint inside the handler function.
3. Trigger the event (click, scroll, keypress).
4. When paused, inspect the `event` object in Scopes and trace from there.

Debugging an async function

Async functions can be debugged the same way as sync ones — the Debugger handles `await` correctly and pauses on the line after the `await` when the Promise resolves. The Call Stack will show the async frame.

Practical Tips

Pause on the exact thrown exception

Enable **Pause on Exceptions** (the hexagon icon in the toolbar). This pauses the moment an error is thrown, before any stack unwinding — the Scopes panel shows exactly what state caused it. Far faster than finding

the error from a Console stack trace.

Edit a value mid-execution to test a fix

When paused, double-click any value in the Scopes panel and change it. Then resume. The program continues with your modified value — a quick way to test whether your hypothesised fix is correct before editing the source.

Step into, then out of, a library function

If you accidentally Step Into a library function, press `Shift + F11` (Step Out) to immediately return to your own code. No need to add a breakpoint or reload.

Use logpoints for loops

Loops that run thousands of times will pause execution thousands of times if you use a regular breakpoint. Use a logpoint instead — the loop runs at full speed and you see all the values in the Console afterwards.

Exercises

1. Open the Debugger (`Ctrl + Shift + Z`) on any website. Browse the file tree on the left. How many JavaScript files are loaded? Click one and read a few lines.

2. Set a line breakpoint inside an event handler (any button's click handler). Click the button. When paused, read the Scopes panel — what variables are in scope?

3. Right-click a line in a loop → **Add Log**. Reproduce the action that runs the loop. Open the Console and observe the logpoint output.

4. Enable **Pause on Exceptions** (hexagon icon). Find a page that produces a JavaScript error in the Console. When the Debugger pauses, read the Call Stack — what is the chain of calls that led to the exception? What values are in scope at the throw point?

5. While paused on a breakpoint, click each step button in turn — Step Over, Step Into, Step Out. Observe how the execution cursor and the Scopes panel change with each step.

6. Find a minified JavaScript file in the file tree (a single very long line). Click **{ }** at the bottom of the editor to pretty-print it. Set a breakpoint on one of the newly-visible lines.

7. Right-click a library file (jQuery, React, Lodash) in the file tree → **Ignore Source**. Set a breakpoint in your own code that calls into the library. Confirm that stepping over the library call no longer enters the library file.

Key Takeaways

- **Line breakpoints** pause execution at a specific line; **conditional breakpoints** pause only when an expression is true.
- **Logpoints** log values to the Console without pausing — ideal for loops and high-frequency events.
- **Pause on Exceptions** catches errors the moment they are thrown, before any stack unwinding.

- **Step Over** moves to the next line; **Step Into** enters a function; **Step Out** returns to the caller.
- **The Call Stack** shows the full chain of calls that led to the current point — click any frame to inspect that context.
- **The Scopes panel** shows every variable in scope, with live values you can edit mid-execution.
- **Blackboxing** keeps the Debugger inside your own code by skipping library and framework files.
- **Source maps** allow debugging against original TypeScript/JSX source even when the browser runs minified code.

What's Next

Lesson 6 covers the **Storage panel** — inspecting and editing cookies, localStorage, sessionStorage, IndexedDB, and Cache Storage directly from DevTools.

LESSON 06 OF 11 · PANEL

The Storage Panel

Overview

The Storage panel gives you a complete view of everything the browser is storing on behalf of the current site: cookies, localStorage, sessionStorage, IndexedDB databases, and the Service Worker Cache. You can read, add, edit, and delete entries directly from DevTools — without writing any code. This makes it indispensable for debugging authentication problems, testing data persistence, inspecting cached assets, and checking that sensitive data is being handled correctly.

Open the Storage panel with `Shift + F9`, or press `F12` and click the **Storage** tab.

Panel Layout

The Storage panel has a tree on the left listing every storage type available for the current origin. Click any node to see its contents in the table on the right.

```
Storage
■■■ Cookies
■ ■■■ https://example.com
■■■ Local Storage
■ ■■■ https://example.com
■■■ Session Storage
■ ■■■ https://example.com
■■■ IndexedDB
■ ■■■ myApp-db
■ ■■■ users (object store)
■■■ Cache Storage
■ ■■■ app-cache-v2
■ ■■■ (cached URLs)
■■■ Extension Storage
■■■ (browser extension data)
```

Cookies

What They Are

Cookies are small key-value pairs the server sends via `Set-Cookie` headers (or JavaScript sets via `document.cookie`). The browser automatically attaches them to every matching request. They are the primary mechanism for session authentication — the session token lives in a cookie so the server knows who is making each request.

Reading Cookies in the Storage Panel

Click **Cookies** → **https://example.com**. The table shows every cookie set for the current origin, with columns for:

Column	Meaning
Name	The cookie's key
Value	The cookie's value (often a session token, JWT, or opaque identifier)
Domain	The hostname scope — which domains the cookie is sent to
Path	The URL path scope — the cookie is only sent for requests under this path
Expires / Max-Age	When the cookie expires. Empty = session cookie (deleted when the browser tab closes)
Size	Total bytes of the name + value
HttpOnly	If checked: JavaScript cannot read this cookie via document.cookie
Secure	If checked: the cookie is only sent over HTTPS connections
SameSite	Controls whether the cookie is sent on cross-site requests
Priority	Chrome/Edge only; not relevant in Firefox

Security Flags

The HttpOnly, Secure, and SameSite flags are the most important columns when auditing a site's cookie security:

HttpOnly

If set, JavaScript cannot access the cookie via `document.cookie`. This is the primary defence against XSS attacks stealing the session token — even if an attacker injects a script, it cannot read the HttpOnly cookie. Session cookies should always have this flag.

Secure

If set, the browser only sends the cookie over HTTPS. Without this flag, the cookie can be intercepted on an HTTP connection. Any cookie containing a session token or sensitive value must have this flag.

SameSite

Controls cross-site request behaviour:

SameSite Value	Behaviour
Strict	Cookie only sent when the request originates from the same site. Never sent on cross-site navigation. Most secure, but breaks links from external sites that land on authenticated pages.

SameSite Value	Behaviour
Lax	Cookie sent on top-level navigations (clicking a link) but not on cross-site subresource requests (images, XHR). The browser default. Good balance of security and usability.
None	Cookie sent on all cross-site requests. Requires the Secure flag to be set. Used for third-party cookies (embedded widgets, payment iframes).

Missing or `None` without `Secure` is a CSRF vulnerability. A session cookie should be `SameSite=Lax` (minimum) or `SameSite=Strict`.

Editing and Deleting Cookies

Edit a cookie value: Double-click any cell in the Value column and type a new value. Press Enter to confirm. The browser immediately uses the new value on subsequent requests — useful for testing what happens with an invalid or expired token.

Delete a single cookie: Select the row and press `Delete`, or right-click → **Delete Cookie**.

Delete all cookies for the origin: Right-click the origin node in the tree → **Delete All** — or click the trash icon in the toolbar.

Add a new cookie: Click in the Name column of the empty row at the bottom of the table and type a new name and value.

Practical Cookie Debugging

Testing an expired session: Delete the session cookie, then try to access a protected page. If the page still loads, the server is not validating the session correctly.

Checking a JWT in a cookie: Copy the Value of a JWT cookie, paste it into jwt.io, and decode it — you can read the payload (sub, exp, roles) without any code. Note that a JWT is signed, not encrypted — the payload is base64-encoded and readable by anyone.

Testing a 401 flow: Edit the session cookie value to an invalid string (e.g. `INVALID`). Make an API call from the page. The server should return 401. If it returns 200, the server is not validating the token.

Local Storage

What It Is

`localStorage` is a persistent key-value store tied to the current origin (scheme + hostname + port). Data persists across browser sessions — closing and reopening the browser, or shutting down the computer, does not clear it. There is no expiry mechanism; data stays until explicitly deleted by JavaScript or by the user clearing their browser data.

Typical uses

- Remembering user preferences (theme, language, sidebar state)
- Caching data to avoid redundant API calls
- Storing a user's progress through a form or wizard
- Authentication tokens (though cookies are generally safer — see the security note below)

Reading and Editing

Click **Local Storage** → **https://example.com**. The table shows two columns: **Key** and **Value**. All values are stored as strings — objects are typically stored as JSON.

Edit a value: Double-click the Value cell and type a new value. The change is applied immediately — the page's JavaScript will read the new value the next time it accesses that key.

Add an entry: Click the empty row at the bottom, type a key, press Tab, type a value.

Delete an entry: Select a row and press `Delete`, or right-click → **Delete Item**.

Clear all localStorage: Right-click the origin node → **Delete All**.

Reading from the Console

You can also read and write localStorage from the Console:

```
// Read all keys and values
for (let i = 0; i < localStorage.length; i++) {
const key = localStorage.key(i);
console.log(key, localStorage.getItem(key));
}

// Read a specific key
localStorage.getItem('theme')

// Set a value
localStorage.setItem('theme', 'dark')

// Delete a key
localStorage.removeItem('theme')
```

Security Note

localStorage is accessible to any JavaScript running on the same origin. This means an XSS attack that injects a script can read everything in localStorage — including authentication tokens. Storing JWT tokens in localStorage (rather than HttpOnly cookies) is a common and widely-criticised pattern. A JWT in localStorage can be stolen by XSS; a JWT in an HttpOnly cookie cannot. For authentication tokens, prefer HttpOnly cookies.

Session Storage

What It Is

`sessionStorage` works identically to `localStorage` in terms of API and the Storage panel interface, with one critical difference: data is scoped to a single browser tab and cleared when that tab is closed. Opening the same page in a new tab gives it a completely separate sessionStorage.

When to Use It

Use Case	localStorage	sessionStorage
Persist across browser restarts	✓	✗
Survive page reload	✓	✓
Survive tab close and reopen	✓	✗
Shared between multiple tabs	✓ (same origin)	✗ (per-tab)
Wizard / multi-step form state	Possible but risky	Ideal
Shopping cart during a session	Possible	Good fit

Session storage is a good fit for data that should not persist between visits — a partially completed form, a temporary filter selection, a one-time authentication flow.

Debugging Session Storage

The key difference to watch for in DevTools: if a user reports that they "lost their data when they opened a new tab", the data was in sessionStorage, not localStorage. The fix is usually to migrate the data to localStorage (if it should persist) or to redesign the flow to not depend on tab continuity.

IndexedDB

What It Is

IndexedDB is a full client-side database built into the browser. Unlike localStorage (which only stores strings), IndexedDB can store structured JavaScript objects, binary data (Blobs, ArrayBuffers), and large volumes of data. It is asynchronous, transactional, and supports indexes for efficient lookups.

Typical uses: offline-capable web apps, caching API responses with complex querying, storing large binary assets (images, audio), and Progressive Web Apps (PWAs).

Exploring IndexedDB in DevTools

Click **IndexedDB** in the left tree. If the page uses IndexedDB, you will see one or more named databases. Expand a database to see its **object stores** (the equivalent of tables). Click an object store to see all its records in the table on the right.

Each record shows its key and value. Objects are displayed as expandable trees — you can navigate into nested properties.

Refreshing the view: IndexedDB data can change while the panel is open. Click the refresh icon (🔄) in the toolbar to reload the current object store's contents.

Deleting a record: Select a row and click the delete icon, or right-click → **Delete**.

Deleting a database: Right-click the database name in the tree → **Delete Database**. This removes all object stores and records. Useful for testing the app's first-run experience.

Practical IndexedDB Debugging

If a PWA is showing stale data, open IndexedDB and read the cached records — compare them to what the live API returns. If the IndexedDB record is older or different, the app's sync logic is not working correctly.

Cache Storage

What It Is

Cache Storage is managed by Service Workers and stores complete HTTP request/response pairs. Unlike browser caching (which the browser controls), Cache Storage is entirely under the application's control — a Service Worker script decides what to cache, when to update it, and what to return from the cache versus the network.

Typical uses: making a web app available offline, pre-caching assets for instant load on repeat visits, and storing API responses for offline access.

Inspecting Cache Storage in DevTools

Click **Cache Storage** in the left tree. If a Service Worker has created any caches, they appear as named entries (e.g. `app-cache-v2`, `runtime-cache`). Click a cache name to see its contents — a table of cached URLs with their status codes, content type, and size.

Click any row to see the full cached response: headers and body. This lets you verify exactly what the Service Worker will return when the network is unavailable.

Deleting a cached entry: Select a row and press `Delete`.

Deleting an entire cache: Right-click the cache name → **Delete**.

Simulating Offline Mode

To test whether your app works offline:

1. Open the **Network panel** and set throttling to **Offline**.
2. Reload the page.
3. If a Service Worker is installed with a working cache strategy, the page loads from Cache Storage. If not, the page fails to load.

Alternatively, use the **Service Workers** section in the Application panel to simulate offline without affecting the Network panel.

Cache vs Browser Cache vs localStorage

Storage	Controlled by	Stores	Max size	Survives close
Browser HTTP cache	Browser	HTTP responses	Browser-managed	Yes
Cache Storage	Service Worker (JavaScript)	HTTP request/response pairs	~50% of disk	Yes
localStorage	JavaScript	Strings	~5–10 MB	Yes
sessionStorage	JavaScript	Strings	~5 MB	No (tab close)
IndexedDB	JavaScript	Structured objects	~50% of disk	Yes
Cookies	Server + JavaScript	Strings	4 KB per cookie	Until expires

---

Storage Limits and Quotas

Browsers enforce storage quotas per origin. You can check how much storage the current origin is using from the Console:

```
const estimate = await navigator.storage.estimate();
console.log(`Used: ${(estimate.usage / 1024 / 1024).toFixed(2)} MB`);
console.log(`Quota: ${(estimate.quota / 1024 / 1024).toFixed(0)} MB`);
```

When storage is full, writes to localStorage, sessionStorage, and IndexedDB throw a `QuotaExceededError`. Cache Storage and IndexedDB may also be evicted by the browser under storage pressure (unless the origin has requested persistent storage via `navigator.storage.persist()`).

Clearing All Storage

The nuclear option: right-click the origin in any storage node → **Delete All** — or open the Console and run:

```
// Clear localStorage and sessionStorage
localStorage.clear();
sessionStorage.clear();

// Clear all cookies (accessible to JavaScript)
document.cookie.split(';').forEach(c => {
document.cookie = c.trim().split('=')[0] +
'=;expires=Thu, 01 Jan 1970 00:00:00 UTC;path=/';
});

// Delete all IndexedDB databases (lists them first)
indexedDB.databases().then(dbs =>
dbs.forEach(db => indexedDB.deleteDatabase(db.name))
);
```

Note: HttpOnly cookies cannot be cleared by JavaScript — they can only be cleared by the server (sending `Set-Cookie` with `Max-Age=0`) or by the user clearing browser data.

Practical Tips

Reproducing a first-visit experience

Clear all storage for the origin (cookies, localStorage, Cache Storage, IndexedDB). Reload. The page should behave exactly as it would for a brand-new visitor — no remembered preferences, no session, no cached data.

Checking whether a PWA is installed correctly

Open Cache Storage and verify the expected cache names are present and contain the right URLs. Open the **Application** panel (in Chrome) or **Service Workers** section to see the Service Worker's status.

Spotting sensitive data in the wrong place

Open localStorage and search for anything that looks like a token, password, or PII. JWT tokens, API keys, and email addresses should not be in localStorage — they belong in HttpOnly cookies or should not be stored client-side at all.

Testing logout

Click the logout button, then check the Storage panel. The session cookie should be gone (or have expired). If any sensitive data remains in localStorage or sessionStorage after logout, it's a bug — subsequent users on a shared machine could access it.

Exercises

1. Open the Storage panel (`Shift + F9`) on a site you are logged into. Find the session cookie. Note which security flags (HttpOnly, Secure, SameSite) are set. Is the session cookie as secure as it should be?

2. Copy the value of any JWT cookie (or localStorage JWT) and paste it into `https://jwt.io`. Read the decoded payload. What claims (sub, exp, role) does it contain?

3. Edit the value of the session cookie to `INVALID`. Make a request from the page (reload, or trigger an API call). Does the server return 401? Restore the original value and confirm you are logged in again.

4. Open localStorage. Are any values stored as JSON? Double-click a value to edit it — change a preference (theme, language) and reload the page. Does the change take effect?

5. Open sessionStorage in one tab and then open the same page in a second tab. Open the Storage panel in both. Add a key in tab one's sessionStorage. Confirm it does not appear in tab two's sessionStorage.

6. Check your storage quota from the Console:

```
navigator.storage.estimate().then(e =>
console.log(`${(e.usage/1024/1024).toFixed(2)} MB used of ${(e.quota/1024/1024).toFixed(0)}
MB`)
);
```

7. If the site uses a Service Worker, open Cache Storage and find a cached URL. Delete one entry and reload the page — does the browser fetch it from the network? (Check the Network panel to confirm.)

Key Takeaways

- **Cookies**: the browser's session mechanism. `HttpOnly` prevents XSS theft; `Secure` enforces HTTPS; `SameSite=Lax` or `Strict` prevents CSRF.
- **localStorage**: persistent, origin-scoped string storage. Accessible to JavaScript — do not store auth tokens here; use HttpOnly cookies instead.
- **sessionStorage**: identical to localStorage but cleared when the tab closes. Ideal for temporary, tab-specific state.
- **IndexedDB**: a full client-side database for structured data and large volumes. Used by PWAs for offline support.
- **Cache Storage**: Service Worker-controlled HTTP response cache. The backbone of offline-capable web apps.
- **Edit values directly** in the Storage panel to test auth flows, simulate expired sessions, or change preferences without touching code.
- **After logout**, check the Storage panel — all session data should be gone. Leftover tokens in any storage type after logout is a bug.

What's Next

Lesson 7 covers the **Performance panel** — recording a performance profile, reading the flame chart, identifying long tasks that block the main thread, and diagnosing slow page loads and janky animations.

LESSON 07 OF 11 · PANEL

The Performance Panel

Overview

The Performance panel records exactly what the browser's main thread was doing at every millisecond during a page load or user interaction — parsing HTML, running JavaScript, calculating styles, laying out elements, painting pixels, and compositing layers. It produces a **flame chart** that shows which functions ran, for how long, and in what order. This is the tool for answering "why is my page slow?" when the answer isn't immediately obvious from the Network panel.

Open the Performance panel with `Ctrl + Shift + P` (macOS: `Cmd + Shift + P`), or press `F12` and click the **Performance** tab.

The Browser Rendering Pipeline

To read a performance profile, you first need to understand what work the browser does to display a frame. Every frame (ideally 60 per second, giving 16.7ms per frame) goes through this pipeline:

```
JavaScript → Style → Layout → Paint → Composite
```

Phase	What happens
JavaScript	JS code runs — event handlers, timers, fetch callbacks, framework renders
Style	The browser recalculates which CSS rules apply to which elements
Layout	The browser calculates the size and position of every element (also called "reflow")
Paint	The browser draws pixels into layers — colours, text, images, borders
Composite	The browser combines the layers and sends the result to the screen

If any phase takes too long, the frame isn't delivered in 16.7ms and the user sees a dropped frame — visible as a stutter or "jank". The Performance panel shows you exactly which phase ate the time budget.

The Main Thread

All JavaScript, style calculation, and layout happen on a single thread — the **main thread**. If your JavaScript runs for 200ms without yielding, the main thread is blocked for that entire time. During this time:

- No user input is processed (clicks, keystrokes feel unresponsive)
- No animations update
- No other JavaScript runs

A task that runs for more than **50ms** is called a **long task** and is highlighted in red in the Performance panel. Long tasks are the most common cause of poor interactivity scores (INP — Interaction to Next Paint).

Recording a Profile

Basic recording

1. Open the Performance panel.
2. Click the **Record** button (⏺) — or press `Ctrl + E`.
3. Perform the action you want to measure — scroll the page, click a button, wait for a load.
4. Click **Stop** (or press `Ctrl + E` again).
5. The profile appears immediately.

Recording a page load

1. Click the **Reload** button in the Performance panel toolbar (the circular arrow icon next to Record). This starts recording and immediately reloads the page, capturing everything from the first network request.

Tips for useful recordings

- Keep recordings short — 3 to 10 seconds. Longer recordings produce massive profiles that are harder to navigate.
- Reproduce only the specific interaction you are diagnosing. If you're debugging a slow button click, record just the moment before and after clicking.
- If you are profiling on a powerful developer machine, the results won't reflect what users on slower devices experience. Use the **CPU throttling** dropdown (4× or 6× slowdown) to simulate mid-range mobile hardware.

Reading the Profile

The profile is displayed in three sections, stacked vertically:

1. The Overview Strip

At the top: a compressed timeline of the entire recording. It shows:

- **FPS bar** (frames per second) — green is good, red means dropped frames
- **CPU bar** — how busy the main thread was (colours correspond to activity types)
- **Network bar** — which requests were in flight

Click and drag on the overview strip to zoom into a time range. This is how you navigate — zoom in on a red FPS section to see what was happening during the drop.

2. The Flame Chart

The main section. Time flows left to right. The vertical axis represents the call stack — functions higher up called the functions below them. A tall column means deeply nested function calls. Width represents time — wider bars took longer.

Each bar is a function call, colour-coded by activity type:

- ■ **Yellow** — JavaScript execution
- ■ **Purple** — Style and Layout (recalculate styles, reflow)
- ■ **Green** — Painting
- ■ **Blue** — HTML parsing
- ■ **Grey** — Idle / Composite

Click any bar to select it. The Details panel below shows the function name, source file, line number, total time, and self time.

Self time vs total time:

- **Total time** — how long this function took including all the functions it called.
- **Self time** — how long this function spent doing its own work, not counting nested calls. A function with high self time is where the actual computation is happening.

3. The Details Panel

Below the flame chart. Depending on what you've selected:

- **Summary** — a pie chart of where time was spent (JS, rendering, painting, idle)
- **Bottom-Up** — functions sorted by self time, highest first. This directly shows which function is most expensive — the best starting point.
- **Call Tree** — the call hierarchy, top-down, showing where time propagates through.
- **Event Log** — every event that fired during the recording, with timestamps.

Long Tasks

A **long task** is any task on the main thread that runs for more than 50ms. In the flame chart they appear with a **red triangle** in the top-right corner of the bar. The 50ms threshold comes from the RAIL model: a user perceives a response as instant if it arrives within 100ms, and a task of 50ms or more consumes more than half that budget.

Finding long tasks

1. Look for red triangles in the flame chart.
2. Click one to see which function(s) are responsible.
3. Check the Details panel → Bottom-Up for the most expensive self-time functions.

Common causes of long tasks

Cause	Description
Large JS bundles	Parsing and compiling a 2 MB bundle on page load blocks the main thread
Synchronous XHR	A synchronous XMLHttpRequest blocks everything until it completes
Unoptimised loops	A loop over thousands of items inside an event handler
Forced layout (thrashing)	Reading a layout property (offsetWidth) immediately after writing a style forces the browser to recalculate layout synchronously
Third-party scripts	Analytics, chat widgets, and ad scripts running expensive code on your page
Large JSON parsing	JSON.parse() on a 5 MB payload blocks the thread while it runs

---

Frame Rate and Animation Performance

The FPS Ruler

The top of the Performance panel shows an FPS (frames per second) ruler. For smooth animation:

- **60 FPS** — ideal. Each frame has 16.7ms.
- **30 FPS** — acceptable for non-interactive content, but animations feel slow.
- **Below 30 FPS** — visibly janky. Users notice.

A dip in the FPS ruler corresponds to a spike in the CPU bar. Zoom into the dip to find the long task that caused it.

Layout Thrashing

One of the most common causes of animation jank. It happens when JavaScript alternates between writing to the DOM and reading layout properties, forcing the browser to recalculate layout synchronously on every read:

```
// Bad — forces layout recalculation on every iteration
elements.forEach(el => {
el.style.width = container.offsetWidth + 'px'; // read, then write
});

// Good — read once, then write in bulk
const width = container.offsetWidth;
elements.forEach(el => {
el.style.width = width + 'px';
});
```

In the Performance panel, layout thrashing appears as repeated purple (Layout) bars inside a JavaScript task — sometimes hundreds of them in rapid succession. The Details panel will name the triggering function.

CSS Properties and the Rendering Pipeline

Not all CSS changes are equally expensive. Properties that only affect compositing are the cheapest — they skip layout and paint entirely:

CSS Property	Triggers	Cost
transform, opacity	Composite only	■ Very cheap — GPU-accelerated
color, background-color, border-color	Paint	■ Medium — repaints the layer
width, height, margin, padding, top, left	Layout + Paint + Composite	■ Expensive — forces full reflow
font-size, line-height	Layout + Paint + Composite	■ Expensive

For animations: always prefer `transform: translateX()` over `left`, and `opacity` over `visibility`. These run on the GPU compositor thread and never block the main thread.

Paint Flashing and Layer Borders

Firefox has two additional visualisation tools accessible from the DevTools settings or the Toolbox options menu:

Paint Flashing

Enable in **Settings** → **Advanced Settings** → **Paint Flashing**. Areas of the page that are repainted on each frame flash green. If you see large areas flashing on every scroll, the page is doing too much repainting — likely because a fixed-position element, animation, or scroll handler is dirtying large areas of the layout.

Layer Borders

Enable in **Settings** → **Advanced Settings** → **Layer Borders**. Shows the boundaries of compositor layers as blue boxes. An element gets its own layer when it has `transform`, `will-change`, `position: fixed`, or certain animations. Layers are good (they let the compositor update just that element without repainting the rest of the page) — but too many layers consume GPU memory.

The Marker Timeline

Some frameworks (React, Vue, Angular) and Performance API calls insert **markers** into the performance timeline. These appear as named vertical lines or labelled regions in the flame chart. You can add your own with:

```
performance.mark('data-fetch-start');
const data = await fetchUsers();
performance.mark('data-fetch-end');
performance.measure('data-fetch', 'data-fetch-start', 'data-fetch-end');
```

The `measure` entry appears in the Performance panel as a named bar spanning the measured duration — exactly like a custom annotation on the timeline.

Core Web Vitals in the Performance Panel

Firefox's Performance panel overlays several important timing markers on the timeline:

Marker	Full name	What it measures
FP	First Paint	When the browser first paints any pixel
FCP	First Contentful Paint	When the first text, image, or canvas is painted
DCL	DOMContentLoaded	When the HTML is parsed and the DOM is ready
L	Load	When all resources (images, scripts) are loaded

These appear as vertical lines on the overview strip. Everything to the left of FCP is what the user sees as a blank screen — minimising this is the goal of performance optimisation for page loads.

Practical Performance Workflow

Diagnosing a slow page load

1. Click **Reload** in the Performance panel toolbar to capture the full load.
2. Look at the FCP marker — how far into the recording does it appear?
3. Zoom into the time before FCP. Look for long yellow (JS) tasks.
4. In Bottom-Up, find the functions with the highest self time. These are your optimisation targets.
5. Check the Network bar — are large scripts blocking parsing? Consider deferring or code-splitting them.

Diagnosing janky animation

1. Start recording, then trigger the animation (scroll, hover, click).
2. Stop and look for red drops in the FPS bar.
3. Zoom into a drop. Look for long tasks in the flame chart.
4. Check if the task contains repeated purple (Layout) bars — likely layout thrashing.
5. Check if the animated properties trigger layout (width, height, top, left) — switch to `transform` if so.

Diagnosing a slow button click

1. Start recording.
2. Click the button.
3. Stop and zoom into the click event.
4. Look for the event handler in the flame chart. How long did it run? What did it call?
5. Check Bottom-Up for the highest self-time function in the handler.

Practical Tips

Always use CPU throttling when profiling for mobile users

Your development machine has far more CPU than a mid-range Android phone. Set throttling to 4× or 6× to get results that reflect real-world conditions. A function that takes 5ms on your MacBook may take 25ms on a user's device.

The Bottom-Up tab is the fastest route to the problem

Don't try to read the entire flame chart from left to right. Open the Details panel → Bottom-Up and sort by Self Time descending. The top entry is the most expensive function — click it to jump to it in the flame chart.

Short recordings are easier to read

A 30-second recording produces thousands of tasks. A 3-second recording focused on the specific interaction is far more actionable.

Record without extensions

Browser extensions run on the same main thread and pollute the performance profile. Record in a Private Window (where extensions are usually disabled) for the cleanest results.

Exercises

1. Open the Performance panel (`Ctrl + Shift + P`) on any page. Click **Record**, scroll the page for 3 seconds, then stop. Look at the FPS bar — are there any red sections (dropped frames)?

2. Click the **Reload** button (⟳) in the Performance panel to capture a full page load. Find the **FCP** marker line. How many milliseconds into the load did the first content appear?

3. Zoom into a yellow (JavaScript) section in the flame chart. Click a bar. Read the function name and file:line in the Details panel. How long did it take?

4. With any recording open, click the **Bottom-Up** tab in the Details panel. Sort by Self Time. Which function is at the top? Is it your code or a library?

5. Enable CPU throttling (4× slowdown) in the Performance panel toolbar. Record the same page load from exercise 2. How much longer did FCP take under throttling?

6. Add performance marks to the Console of any page and see them in a recording:

```
performance.mark('start');
// do some work
for (let i = 0; i < 1000000; i++) {}
performance.mark('end');
performance.measure('my-work', 'start', 'end');
```

Then record a profile and look for the `my-work` measure in the timeline.

7. Find an animated element on any page (a CSS animation, a hover transition). Record while triggering it. Look for the animation in the flame chart. Is it green (paint/composite) or purple (layout)? A purple animation is a candidate for switching to `transform`/`opacity`.

Key Takeaways

- The browser renders in five stages: **JavaScript** → **Style** → **Layout** → **Paint** → **Composite**. Each must fit in 16.7ms for 60 FPS.
- **Long tasks** (>50ms, marked with a red triangle) block the main thread and make the page feel unresponsive.
- **The flame chart** shows what ran and for how long — width = time, height = call depth.
- **Bottom-Up** sorted by Self Time is the fastest way to find which function is most expensive.
- **Layout thrashing** (reading layout properties after writing styles) causes repeated forced reflows — read once, then write in bulk.
- `transform` **and** `opacity` are GPU-accelerated and never trigger layout — use them for animations instead of `width`, `top`, `left`.
- **CPU throttling** is essential for realistic mobile profiling — your dev machine is far faster than users' devices.
- **Performance marks** let you annotate your own timeline to measure specific operations.

What's Next

Lesson 8 is the first scenario lesson — **Inspecting Authentication** — where we apply the tools from lessons 1–7 to a real task: determining what authentication mechanism a website uses, reading the tokens it sends, and understanding the flow from login to API call.

LESSON 08 OF 11 · SCENARIO

Scenario: Inspecting Authentication

Overview

This is the first scenario lesson. Rather than introducing a new panel, we apply the tools from lessons 1–7 to a single real-world task: **determining exactly how a website authenticates its users**. This includes identifying the authentication mechanism in use, reading the tokens that are sent, tracing the full journey from the login form submission to a protected API call, and diagnosing common authentication failures.

By the end of this lesson you will be able to sit down in front of any web application and answer:

- What authentication mechanism does it use (session cookie, JWT, OAuth, API key)?
- What token is being sent, and where is it stored?
- Is it configured securely?
- Why is the authentication failing?

Step 1 — Open the Right Panels

Before you do anything else, open two panels simultaneously:

1. Press `F12` to open DevTools.
2. Click the **Network** tab.
3. Press `Escape` to open the **Console** drawer at the bottom.

Now you can watch network requests in the top panel and run JavaScript queries in the Console below — without switching tabs.

Also enable **Persist Logs** in the Network panel toolbar. Authentication often involves page redirects, and without Persist Logs you will lose the login request the moment the redirect fires.

Step 2 — Identify the Authentication Mechanism

Not all websites use the same authentication approach. The first task is to determine which one is in use. Here is how to identify each:

Session Cookie Authentication (traditional)

What it looks like:

- The login request (`POST /login` or `POST /auth/signin`) returns a `Set-Cookie` header in the response.

- Subsequent API requests carry a `Cookie` header automatically.
- No `Authorization` header appears on API requests.

How to find it in DevTools:

1. Filter the Network panel by `XHR/Fetch`.
2. Click the login request.
3. Check the **Response Headers** for `Set-Cookie`.
4. Check the **Storage panel** (Cookies) — you should see the new session cookie.
5. On a subsequent authenticated request, check the **Request Headers** for a `Cookie` header.

What a secure session cookie looks like:

```
Set-Cookie: session_id=abc123; HttpOnly; Secure; SameSite=Lax; Path=/; Max-Age=86400
```

All four flags — `HttpOnly`, `Secure`, `SameSite`, and an explicit expiry — should be present.

JWT Bearer Token Authentication

What it looks like:

- The login request returns a JSON body containing a token field (`access_token`, `token`, `jwt`).
- Subsequent API requests include an `Authorization: Bearer <token>` header.
- The token is stored in either localStorage or a cookie.

How to find it in DevTools:

1. Filter the Network panel by `XHR/Fetch`.
2. Click the login request → **Response** tab.
3. Look for a JSON body containing a token:

```
{
"access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI0MiIsInJvbGUiOiJhZG1pbiIs
ImV4cCI6MTczMDAwMDAwMH0.abc123",
"token_type": "Bearer",
"expires_in": 3600
}
```

4. Click a subsequent API call (e.g. `GET /api/user/profile`) → **Request Headers**.
5. Find `Authorization: Bearer eyJhbGci…` — this is the token being sent.
6. Check the Storage panel → localStorage to see where the token is stored.

Decoding the JWT:

A JWT has three parts separated by dots: `header.payload.signature`. The header and payload are base64url-encoded (not encrypted). Decode them directly in the Console:

```
// Decode the payload of any JWT
const token = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI0MiIsInJvbGUiOiJhZG1pbiIsIm
V4cCI6MTczMDAwMDAwMH0.abc123';
const payload = JSON.parse(atob(token.split('.')[1]));
console.table([payload]);
```

Output:

```
┌──────────┬──────────────┐
│ sub      │ "42"         │
│ role     │ "admin"      │
│ exp      │ 1730000000   │
└──────────┴──────────────┘
```

Convert `exp` to a human date:

```
new Date(payload.exp * 1000).toLocaleString();
// → "27/10/2024, 14:13:20"
```

OAuth 2.0 / OpenID Connect

What it looks like:

- Clicking "Sign in with Google / GitHub / Microsoft" redirects the browser to a third-party URL (`accounts.google.com`, `github.com/login/oauth/authorize`).
- After the user approves, the third party redirects back with a `code` parameter in the URL.
- The application exchanges the code for tokens with a backend call.

How to find it in DevTools:

1. Enable **Persist Logs** in the Network panel (critical — there will be redirects).
2. Click the "Sign in with…" button.
3. Watch the Network panel for:
- A redirect to the provider's URL (`/oauth/authorize?client_id=…&redirect_uri=…`)
- A redirect back to your app with `?code=…` in the URL
- A backend call to exchange the code for tokens (`POST /oauth/token` or similar)
4. The final token exchange response will contain the access token (and often a refresh token).

The `redirect_uri` parameter in the initial request is important — it tells the provider where to send the user back. A misconfigured `redirect_uri` is a common cause of OAuth failures.

API Key Authentication

What it looks like:

- Requests carry a static key in one of: an `Authorization` header (`Authorization: ApiKey abc123`), a custom header (`X-API-Key: abc123`), or a query parameter (`?api_key=abc123`).
- There is no login flow — the key is configured once and used on every request.

How to find it in DevTools:

1. Filter the Network panel by `XHR/Fetch`.
2. Click any API request → **Request Headers**.
3. Look for an `Authorization`, `X-Api-Key`, `X-Auth-Token`, or similar header.
4. Alternatively check the URL in the request list for a `key=` or `api_key=` query parameter.

API keys in URL query parameters are a security concern — the key appears in server logs, browser history, and HTTP Referer headers. Keys should always be in headers.

Step 3 — Trace the Full Login Flow

With Persist Logs enabled, do a fresh login and watch the entire sequence in the Network panel. Here is what a typical JWT-based login looks like, request by request:

#	Method	URL	What's happening
1	POST	/auth/login	Username and password sent in the request body
2	← 200	/auth/login	Server validates credentials, returns access_token and refresh_token in the response body
3	JavaScript stores the token in localStorage or a cookie		
4	GET	/api/user/profile	First authenticated request — Authorization: Bearer <token> header is present
5	GET	/api/dashboard/data	Second authenticated request — same Bearer header

For session-cookie authentication, step 2 returns a `Set-Cookie` header instead of a JSON token, and steps 4–5 carry a `Cookie` header instead of `Authorization`.

Reading the login request body:

Click the login request → **Request** tab. The body should contain the credentials. Common formats:

```
// JSON body (Content-Type: application/json)
{ "email": "alice@example.com", "password": "••••••••" }

// Form-encoded body (Content-Type: application/x-www-form-urlencoded)
email=alice%40example.com&password=••••••••
```

If `Content-Type` is `application/json`, the **Request** tab shows a formatted JSON tree. If it is form-encoded, it shows key-value pairs.

Step 4 — Check Security Configuration

Once you know what tokens are in use and where they are stored, audit the security configuration:

Cookie-based auth checklist

Open the Storage panel → Cookies and check each session cookie:

Check	Secure configuration	Risk if missing
HttpOnly flag	✓ must be set	XSS can steal the token via document.cookie
Secure flag	✓ must be set	Token sent over HTTP — interceptable on public Wi-Fi
SameSite=Lax or Strict	✓ should be set	CSRF — attacker can trigger authenticated requests from another site
Expiry set	✓ should be explicit	Session never expires — tokens valid indefinitely
Token is opaque (not a JWT)	✓ preferred	JWT session tokens are large and expose claims to clients

JWT-in-localStorage checklist

Open the Storage panel → localStorage and check:

Check	Issue
Token is in localStorage at all	Readable by any JS — XSS can steal it
Token expiry (exp claim)	Decode the JWT in the Console — is it reasonable (hours, not years)?
Token contains sensitive claims	The payload is readable — does it expose PII (email, phone number)?
Refresh token also in localStorage	Refresh tokens have long lifetimes — extra dangerous if stolen

Reading a JWT's expiry from the Console

```
// Paste the token from localStorage or the Authorization header
const token = localStorage.getItem('access_token');
const { exp, sub, role } = JSON.parse(atob(token.split('.')[1]));
console.log('User:', sub);
console.log('Role:', role);
console.log('Expires:', new Date(exp * 1000).toLocaleString());
console.log('Expired?', Date.now() > exp * 1000);
```

Step 5 — Diagnose Authentication Failures

When something goes wrong with authentication, the Network panel tells you exactly what and where.

401 Unauthorized

The server received a request but did not find valid credentials.

Diagnosis steps:

1. Click the 401 request → **Request Headers**.
2. Is `Authorization` present? If not → the client is not sending credentials. Check the JavaScript that should add the header, or check that the token exists in localStorage/cookies.
3. Is `Authorization` present but still 401? → The token is invalid or expired. Decode the JWT `exp` claim in the Console. Check if the token was revoked server-side.
4. Check the **Response** tab for an error message — most APIs return a JSON body like `{"error": "token_expired"}` or `{"error": "invalid_signature"}`.

403 Forbidden

The server recognised the credentials but denied access.

Diagnosis steps:

1. Click the 403 request → decode the JWT in the Console.
2. Check the `role` or `scope` claim — does the user have the required permission?
3. Check the Response body — the server often explains why (`{"error": "insufficient_scope"}`).

Redirect loop on login

The browser keeps bouncing between the login page and a protected page.

Diagnosis steps:

1. With Persist Logs on, reload and watch the Network panel.
2. Look for repeated 302 redirects between two URLs.
3. Check the `Location` response header on each redirect — where is the server sending the browser?
4. Check the Cookie storage — is the session cookie being set correctly after login? If the redirect lands on the page before the cookie is saved, the server sees an unauthenticated request and redirects back to login.

CORS error blocking an authenticated request

Appears in the Console as a red error and in the Network panel as a request with no response.

Diagnosis steps:

1. Open the Console — read the full CORS error message. It names the missing header.
2. Click the failed request in the Network panel → **Response Headers**.
3. Check for `Access-Control-Allow-Origin` — is it present? Does it match the page's origin?

4. Check for `Access-Control-Allow-Headers` — does it include `Authorization`? If not, the server is not allowing the auth header cross-origin.
5. Look for a `OPTIONS` preflight request immediately before the failed request — what did it return?

A common misconfiguration: `Access-Control-Allow-Origin: *` is set but `Access-Control-Allow-Headers` does not include `Authorization`. The wildcard origin allows the request but the auth header is stripped.

Step 6 — Simulate and Test

Once you understand the auth flow, test edge cases directly from DevTools:

Test token expiry

1. Decode the JWT `exp` in the Console — find out when it expires.
2. If it expires soon, wait. Or:
3. Go to Storage → localStorage → edit the `access_token` value, replacing the last few characters to corrupt the signature.
4. Trigger an API call. The server should return 401.
5. Does the application handle the 401 gracefully — show a login prompt, redirect to login, or display an error message?

Test logout completeness

1. Log in and note what appears in Cookies and localStorage.
2. Click the logout button.
3. Open the Storage panel — are all session tokens and cookies gone?
4. Manually type a protected URL in the address bar. Are you redirected to login?

Test cross-tab isolation

1. Log in in tab 1.
2. Open a new tab on the same site. Are you also logged in?
3. Log out in tab 1.
4. Refresh tab 2. Are you still logged in? (If yes, the logout endpoint did not invalidate the server-side session — only the local cookie was cleared.)

Practical Tips

The Network filter `Authorization` finds authenticated requests instantly

In the Network panel, press `Ctrl + F` to open the network search box and type `Authorization`. This searches inside request headers and shows every request that carried an auth token — without clicking each one.

`$0` lets you extract tokens from hidden form inputs

Some apps store CSRF tokens in hidden `<input>` fields. Select the form in the Inspector, switch to the Console and run `$0.querySelector('[name=_token]').value` to read it.

Check the OPTIONS preflight before blaming CORS on the API

Before assuming the server is wrong, check whether a CORS preflight (`OPTIONS`) request was made and what it returned. A missing preflight often means the client is not configured for CORS at all.

Copying a token to test with curl or Postman

In the Network panel, right-click any authenticated API request → **Copy as cURL**. This copies a `curl` command with all the exact headers — including the `Authorization` token — ready to paste into a terminal or import into Postman for further testing.

Exercises

1. Open the Network panel on a site you are logged into. Filter by `XHR/Fetch`. Click any API request → **Request Headers**. Which authentication mechanism is in use — Cookie, Bearer token, or something else?

2. If the site uses JWT: copy the `Authorization` header value (the `eyJ…` part). Run the following in the Console to decode it:

```
const t = 'PASTE_TOKEN_HERE';
console.table(JSON.parse(atob(t.split('.')[1])));
```

What claims does the payload contain? When does the token expire?

3. Find the login request in the Network panel. Click it → **Request** tab. What format is the login body (JSON or form-encoded)? Is the password visible in plain text?

4. Open Storage → Cookies after logging in. Does the session cookie have `HttpOnly`, `Secure`, and `SameSite` set? Note any flags that are missing.

5. Edit the session cookie or localStorage token to an invalid value. Trigger an authenticated request. Does the server return 401? Does the application handle the error gracefully?

6. Log out and check the Storage panel. Are all auth tokens gone? Navigate directly to a protected URL (e.g. `/dashboard`). Are you redirected to login?

7. Right-click any authenticated API request → **Copy as cURL**. Paste the result into a terminal or note it — can you see the exact token that was sent?

Key Takeaways

- **Session cookies** are identified by a `Set-Cookie` response header on login and a `Cookie` request header on subsequent calls.

- **JWT Bearer tokens** appear in the login response body and on subsequent calls as `Authorization: Bearer <token>`.
- **Enable Persist Logs** before any login flow — the login request is lost otherwise when the page redirects.
- **Decode any JWT in the Console** with `JSON.parse(atob(token.split('.')[1]))` — no third-party tool needed.
- **401** means credentials are absent or invalid; **403** means credentials are valid but access is denied — check the role/scope claim.
- **CORS errors** on authenticated requests usually mean `Access-Control-Allow-Headers` is missing `Authorization` — a server-side misconfiguration.
- **"Copy as cURL"** (right-click any request) gives you an instantly replayable command with all headers — invaluable for testing APIs outside the browser.
- After logout, **verify in the Storage panel** that all tokens are truly gone — leftover session data after logout is a security bug.

What's Next

Lesson 9 covers the second scenario — **Troubleshooting Client Access** — diagnosing why a user cannot reach or use your web application: CORS errors, mixed content, DNS failures, firewall blocks, TLS issues, and proxy-related problems.

LESSON 09 OF 11 · SCENARIO

Scenario: Troubleshooting Client Access

Overview

This lesson applies everything from lessons 1–8 to one of the most common real-world situations a web developer faces: **a user cannot access or use your web application**. The failure could be on the network (DNS, firewall, TLS), in the browser (CORS, mixed content), or in the environment (proxy, VPN, corporate filter). Each failure type leaves a distinct fingerprint in DevTools.

By the end of this lesson you will be able to diagnose:

- **CORS errors** — why the browser is blocking cross-origin requests
- **Mixed content** — why an HTTPS page is refusing to load HTTP resources
- **TLS / certificate errors** — why the browser shows a security warning
- **DNS failures** — why the domain cannot be resolved
- **Network/firewall blocks** — why requests time out or are refused
- **Proxy issues** — how an intermediary is interfering with requests

The Diagnostic Mindset

Before touching DevTools, ask two questions:

1. **What exactly is failing?** A blank page, a missing image, a broken API call, and a login redirect loop are four different problems that need four different tools.
2. **Is it the same for everyone?** A problem that only affects one user on a corporate network is likely a firewall or proxy issue, not a bug in your code.

Open DevTools with `F12`, go to the **Network** panel, and **check the Console** (press `Escape` to open the Console drawer). Most access failures generate either a red network entry or a red Console error — often both.

Step 1 — Read the Console Error

The Console is the fastest diagnostic starting point. Access errors are almost always logged there. Common messages and what they mean:

Console error message	Diagnosis
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at https://api.example.com/…	CORS — the server did not send the correct Access-Control-Allow-Origin header
Mixed Content: The page at 'https://…' was loaded over HTTPS, but requested an insecure resource 'http://…'	Mixed content — an HTTP resource on an HTTPS page
NS_ERROR_UNKNOWN_HOST	DNS failure — the domain could not be resolved
NS_ERROR_CONNECTION_REFUSED	The host is reachable but no service is listening on that port
NS_ERROR_NET_TIMEOUT	The connection timed out — firewall, slow server, or overloaded upstream
SEC_ERROR_UNKNOWN_ISSUER / SSL_ERROR_BAD_CERT_DOMAIN	TLS/certificate error — invalid or mismatched certificate
net::ERR_TUNNEL_CONNECTION_FAILED	Proxy failure — the CONNECT tunnel could not be established

Write the error message down. It directly names the problem category. Then open the Network panel for the detail.

Step 2 — CORS Errors

CORS (Cross-Origin Resource Sharing) is a browser security policy that prevents pages at one origin (`https://app.example.com`) from reading responses from a different origin (`https://api.other.com`) unless the server explicitly allows it.

What it looks like in DevTools

Console:

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource
at https://api.other.com/data. (Reason: CORS header 'Access-Control-Allow-Origin' missing).
```

Network panel:

- The request appears in the list, usually with a status of `--` or `0` (no response was received by the page).
- The request *was* sent and a response *was* received by the browser — but the browser suppressed the response from the page's JavaScript.

How to investigate

1. Click the blocked request in the Network panel.
2. Click the **Response Headers** tab.
3. Look for `Access-Control-Allow-Origin`. Is it present? Does it match the page's origin?

If the header is **missing** entirely → the server has no CORS configuration for this endpoint. This is a back-end fix.

If the header is present but wrong (e.g. `Access-Control-Allow-Origin: https://staging.example.com` but the page is on `https://app.example.com`) → the server's allowed-origins list needs updating.

4. Check whether there is an **OPTIONS preflight** request immediately before the blocked request. The browser sends a preflight to ask the server what it allows. If the preflight failed, the actual request is never sent.

Click the OPTIONS request → **Response Headers**:

```
Access-Control-Allow-Origin: https://app.example.com
Access-Control-Allow-Methods: GET, POST, DELETE
Access-Control-Allow-Headers: Content-Type, Authorization
Access-Control-Max-Age: 3600
```

If `Access-Control-Allow-Headers` is missing `Authorization`, requests with a Bearer token will be blocked even if the origin header is correct.

CORS and `credentials: 'include'`

When a fetch is made with `credentials: 'include'` (to send cookies cross-origin), the server must respond with:

- `Access-Control-Allow-Origin: https://app.example.com` (a specific origin, **not** `*`)
- `Access-Control-Allow-Credentials: true`

A wildcard `Access-Control-Allow-Origin: *` combined with `credentials: 'include'` is explicitly forbidden by the browser and causes a CORS error even if the origin is technically allowed.

Console shortcut — read the reason code

Firefox appends a reason code to CORS error messages in the Console:

Reason code	What it means
CORS header 'Access-Control-Allow-Origin' missing	No header at all
CORS request did not succeed	Network error before the response — server may be down
CORS header 'Access-Control-Allow-Origin' does not match	Wrong origin in the header
Credential is not supported if the CORS header 'Access-Control-Allow-Origin' is '*'	Wildcard + credentials conflict
CORS preflight response did not succeed	OPTIONS request returned a non-2xx status
CORS header 'Access-Control-Allow-Headers' is missing	Custom header (e.g. Authorization) not listed

---

Step 3 — Mixed Content

HTTPS pages must load all sub-resources (scripts, images, XHR requests, WebSockets) over HTTPS. A resource loaded over HTTP on an HTTPS page is called **mixed content**. Browsers block **active** mixed

content (scripts, iframes, fetch requests) entirely. Passive mixed content (images, audio) is sometimes allowed but flagged as a warning.

What it looks like in DevTools

Console:

```
Mixed Content: The page at 'https://shop.example.com' was loaded over a secure connection, but is attempting to access 'http://cdn.example.com/logo.png'. This content should be served over HTTPS.
```

Network panel:

- Blocked active mixed content: the request shows status `--` (blocked before it was sent).
- Passive mixed content (images): the request succeeds but the lock icon in the address bar breaks.

How to investigate

1. Search the Network panel: press `Ctrl + F` and type `http://`. This searches response bodies and request URLs — look for any result that contains an HTTP URL on an HTTPS page.
2. Filter the Console by "Error" and look for "Mixed Content" entries.
3. Each Console error names the exact HTTP URL that caused the block.

Common causes

- A hardcoded `http://` URL in HTML, CSS, or JavaScript that was written before the site moved to HTTPS.
- A third-party embed (video player, map widget, analytics script) that still serves over HTTP.
- A CSS `background-image: url('http://…')` in a stylesheet.
- An API endpoint URL stored in a config file that was not updated when the site moved to HTTPS.

Fix and verify

Change the offending `http://` URL to `https://` (or to a protocol-relative URL `//cdn.example.com/logo.png`). After the fix, reload with DevTools open and confirm no mixed content warnings appear in the Console.

Step 4 — TLS / Certificate Errors

A TLS error means the browser could not establish a secure connection — because the certificate is invalid, expired, self-signed, or doesn't match the domain.

Types of TLS errors

Error code	Meaning
SEC_ERROR_EXPIRED_CERTIFICATE	Certificate is past its Not After date

Error code	Meaning
SEC_ERROR_UNKNOWN_ISSUER	Certificate was issued by a CA the browser doesn't trust (self-signed, private CA)
SSL_ERROR_BAD_CERT_DOMAIN	Certificate's Common Name or SAN doesn't match the domain in the URL
SSL_ERROR_RX_RECORD_TOO_LONG	Usually means HTTP traffic was sent to an HTTPS port (misconfigured server)
MOZILLA_PKIX_ERROR_ADDITIONAL_POLICY_CONSTRAINT_FAILED	Certificate violates a browser policy (too old, using deprecated algorithm)

Investigating with DevTools

TLS errors often prevent the page from loading at all — so DevTools can't show you much. Use the **lock icon** in the address bar instead:

1. Click the lock icon → **Connection Secure** (or the warning icon if there's a problem).
2. Click **More Information**.
3. The Security tab shows the certificate details: issuer, validity period, subject alternative names (SANs).

For TLS errors that only affect specific sub-resources (e.g. an API on a different domain):

1. Open the Network panel.
2. Find the failed request — it shows a red status with no response code.
3. Click it → the **Security** tab in the request details shows the certificate error.

Self-signed certificates in development

During local development, self-signed certificates are common. In Firefox, you can temporarily bypass the warning by clicking **Advanced** → **Accept the Risk and Continue** — but this should never be done in production. If your development stack uses a local CA (like `mkcert`), import the CA root certificate into Firefox via **Settings** → **Privacy & Security** → **Certificates** → **View Certificates** → **Import**.

Step 5 — DNS Failures

DNS failure means the browser could not resolve the domain name to an IP address. The page shows Firefox's "Server not found" error, and the Console shows `NS_ERROR_UNKNOWN_HOST`.

What it looks like in DevTools

Network panel:

- The request shows status `--` with no response.
- Hover over the entry — the tooltip may say "DNS resolution failed".
- The **Timings** tab shows DNS Lookup at the top, with no other phases — the request never got past name resolution.

Investigating DNS issues

DNS failures are usually environmental rather than a bug in the website itself. Likely causes:

- The user is on a corporate network with DNS filtering (the domain is blocked by policy).
- The domain hasn't propagated yet (new domain, recent DNS change — TTL not expired).
- The user's DNS server is down or misconfigured.
- The domain has genuinely expired.

Verify the DNS from DevTools:

1. Open the Console.
2. Check the Network panel Timings for the failing request — a DNS phase that takes very long (> 5 seconds) then fails indicates the resolver is running but the name doesn't exist.
3. A Timings panel showing 0ms DNS then immediate failure usually means the name was already known-bad (cached NXDOMAIN).

Quick verification from the Console:

```
// Attempt a fetch to see the raw network error
fetch('https://api.example.com/health')
.then(r => console.log('OK', r.status))
.catch(e => console.error('Failed:', e.message));
```

A DNS failure produces `Failed: NetworkError when attempting to fetch resource.` — the same error as a firewall block. You can't distinguish them from inside the browser alone; you need a terminal command like `nslookup` or `dig` from the same network.

Step 6 — Network / Firewall Blocks

A firewall block or port block looks similar to DNS failure in DevTools — requests time out or are refused — but the domain resolves successfully. The Network panel shows requests that take a long time and then fail with status `--`.

Identifying the pattern

Connection refused (`NS_ERROR_CONNECTION_REFUSED`):

- The host IP was reached but the port is closed or nothing is listening.
- Timing: very fast failure (sub-millisecond) — the host immediately sent a TCP RST.
- Common cause: wrong port number, service not running, port blocked at the firewall.

Connection timeout (`NS_ERROR_NET_TIMEOUT`):

- DNS succeeded but no TCP connection could be established.
- Timing: slow failure — after 30–120 seconds of waiting with no response.
- Common cause: firewall dropping packets without sending a RST; VPN routing issue; server overloaded.

How to tell them apart in the Timings tab:

- Click the failed request → **Timings**.
- If only "DNS Lookup" and "Connecting" are shown, and "Connecting" took 30+ seconds → timeout.
- If the entire timing is 0ms → refused (the RST came immediately).

Checking throttling settings

Before assuming a network block, check: is the Network panel's throttling set to something very slow?

- Look at the toolbar — there is a throttling dropdown (default: "No throttling").
- If it's set to "GPRS" or "Regular 3G", that may explain slow requests.

Corporate network and firewall diagnosis

If the user is on a corporate network:

1. Ask whether other users on the same network have the same problem.
2. Check the blocked domain against the company's firewall policy list.
3. Test from a different network (mobile hotspot) — if it works, the firewall is the cause.

Step 7 — Proxy Issues

A proxy sits between the browser and the internet, inspecting or modifying HTTP/HTTPS traffic. Corporate environments commonly use proxies for security, compliance, or caching. Proxy issues cause a wide range of errors.

Common proxy error symptoms

Symptom	What the proxy is doing
407 Proxy Authentication Required	Proxy requires credentials — the browser didn't send them
502 Bad Gateway from an internal IP	Proxy received an invalid response from the upstream server
504 Gateway Timeout from an internal IP	Proxy could not reach the upstream server
ERR_TUNNEL_CONNECTION_FAILED	The proxy's CONNECT tunnel for HTTPS could not be established
Unexpected response modifications	Proxy is injecting scripts, modifying headers, or stripping HTTPS

Identifying proxy involvement

In the Network panel, click any request → **Headers** tab. Look for:

```
Via: 1.1 corporate-proxy-01 (squid/5.3)
X-Forwarded-For: 10.0.0.45
X-Cache: HIT from corporate-proxy-01
```

The presence of `Via`, `X-Forwarded-For`, or `X-Cache` headers in the response proves a proxy is in the chain.

HTTPS interception (SSL inspection)

Some corporate proxies perform **HTTPS inspection** — they terminate the TLS connection at the proxy, inspect the plaintext, then re-encrypt it with their own certificate before forwarding. The browser sees the proxy's certificate instead of the server's real certificate.

How to detect it:

1. Click the lock icon → **More Information** → **Security** tab (or check the Security tab in a Network request's detail pane).
2. Look at the **Issued By** field. If it says something like "Corporate SSL Inspector" or "Zscaler Root CA" instead of "Let's Encrypt" or "DigiCert", the connection is being inspected.

This is not necessarily malicious (it's common in corporate environments), but it breaks certificate pinning and may cause unexpected errors if the proxy's root CA is not trusted by the browser.

Proxy bypass test

To test whether the proxy is the cause:

1. Configure Firefox to use no proxy: **Settings** → **General** → **Network Settings** → **No proxy**.
2. Reload the failing page.
3. If the error changes or disappears, the proxy was involved.
4. Note: bypassing a corporate proxy may violate network policy — confirm with your IT department.

Putting It All Together — A Diagnostic Flowchart

When a user reports they cannot access your application:

```
1. Open DevTools → Console
■■ Is there a red error message?
■■ CORS header missing → Step 2
■■ Mixed Content → Step 3
■■ SEC_ERROR / SSL_ERROR → Step 4
■■ NS_ERROR_UNKNOWN_HOST → Step 5
■■ NS_ERROR_CONNECTION_REFUSED / NET_TIMEOUT → Step 6
■■ 407 / ERR_TUNNEL → Step 7

2. Open Network panel → find the failing request
■■ Check Status: 4xx/5xx = server responded
■■ Check Status: -- = no response (network/browser block)
■■ Click request → Timings tab
■■ Long DNS = DNS failure
■■ Long Connecting = firewall/proxy timeout
■■ Short all-zero = connection refused

3. Check Headers
■■ Missing CORS headers → back-end configuration
■■ Via / X-Cache in response → proxy in chain
■■ Unexpected 407 → proxy auth required
```

Practical Tips

Check the Console first, always

The Console error message almost always names the exact problem. Don't spend five minutes in the Network panel if the Console has already told you `CORS header 'Access-Control-Allow-Origin' missing`.

`Ctrl + F` in the Network panel searches response bodies

Type `http://` to find all plaintext HTTP URLs in responses — useful for hunting down mixed content sources buried in a CSS file or a JSON response.

Use the Throttling setting to simulate poor connectivity

Set the Network panel to "Regular 3G" and reload. If the page now times out but normally loads, the page is right at the edge of what a slow connection can handle — not a firewall issue.

Block specific URLs to test fallback behaviour

Right-click any request in the Network panel → **Block URL**. This prevents the browser from making that request. Use it to test: what happens when a CDN is unreachable? Does the page degrade gracefully?

Check the request from a different network

If the error is `NS_ERROR_UNKNOWN_HOST` or `NS_ERROR_NET_TIMEOUT`, open a mobile hotspot and try again from the same computer. If it works over mobile data, the corporate network or DNS is the cause.

Save a HAR file before clearing the log

If you have captured a complex failure trace across many requests, right-click the request list → **Save All as HAR** before clearing. This preserves the entire session for sharing with a colleague or comparing against a

working trace.

Exercises

1. Open `https://developer.mozilla.org` in Firefox with DevTools open. In the Console, type:

```
fetch('http://example.com/api/test');
```

Read the Console error. Is this a CORS error, a mixed content error, or both? Why?

2. In the Network panel, type `http://` in the search box (`Ctrl + F`). Does the response to any request on the current page contain an HTTP URL embedded in it?

3. Visit any HTTPS site and click the lock icon in the address bar → **More Information** → **Security** tab. Who issued the TLS certificate? When does it expire?

4. In the Network panel toolbar, set throttling to **GPRS**. Reload any resource-heavy page. Watch the waterfall. How does the timing change compared to no throttling?

5. Right-click any request in the Network panel → **Block URL**. Reload the page. What happens to the page when that request cannot be made? Does it fail silently or show an error?

6. Visit a page that makes XHR/Fetch API calls. Filter the Network panel to XHR/Fetch. Click one of the API requests → **Response Headers**. Is `Access-Control-Allow-Origin` present? What is its value?

7. Simulate a DNS failure by adding a fake entry to your `/etc/hosts` file (on Windows: `C:\Windows\System32\drivers\etc\hosts`) pointing `bad.example.com` to `0.0.0.0`, then try `fetch('https://bad.example.com')` in the Console. What does the Timings panel show for this request?

Key Takeaways

- **Read the Console first** — the error message names the problem category. Don't investigate blindly.
- **CORS errors** are a browser enforcement of the Same-Origin Policy — the server must send `Access-Control-Allow-Origin` and, for credentialed requests, `Access-Control-Allow-Credentials: true`.
- **Mixed content** is any HTTP resource on an HTTPS page — active mixed content is blocked, passive content generates warnings. Fix by changing URLs to `https://`.
- **TLS errors** are visible via the lock icon → More Information, and in the Security tab of Network request details.
- **DNS failures** show as `NS_ERROR_UNKNOWN_HOST` in the Console; the Timings tab shows only a failed DNS phase.
- **Firewall blocks** time out slowly; **connection refused** fails immediately — the Timings tab distinguishes them.
- **Proxy involvement** is revealed by `Via`, `X-Cache`, or `X-Forwarded-For` headers in responses.

- **Block URL** (right-click any request) lets you test what happens when a resource is unavailable — useful for testing resilience.

What's Next

Lesson 10 covers the third scenario — **Debugging a Slow or Broken Page** — diagnosing why a page renders incorrectly, behaves unexpectedly, or loads slowly. This ties together the Inspector, Console, Debugger, Network, and Performance panels into a single diagnostic workflow.

LESSON 10 OF 11 · SCENARIO

Scenario: Debugging a Slow or Broken Page

Overview

This is the broadest scenario lesson. A page can be "broken" in many ways — it shows wrong content, crashes after a click, takes 10 seconds to load, or freezes the browser during a scroll. Each failure type has a natural home in a specific DevTools panel, but real bugs often span multiple panels. This lesson teaches a systematic triage process: start broad, narrow down, then dig in with the right tool.

By the end of this lesson you will be able to diagnose:

- **Visual / layout bugs** — elements missing, overlapping, or the wrong size
- **JavaScript errors** — uncaught exceptions, undefined references, wrong values
- **Slow page loads** — large resources, render-blocking scripts, slow APIs
- **Runtime performance** — jank, frozen interactions, layout thrashing

The Triage Order

Resist the urge to immediately open the Debugger and start stepping through code. Start at the highest level and work inward:

```
1. Console → Is there an uncaught error? (10 seconds)
2. Network → Did all resources load? Are any requests slow? (30 seconds)
3. Inspector → Is the DOM what you expect? Are styles applied correctly? (1 minute)
4. Debugger → Where exactly in the code does it go wrong? (5–20 minutes)
5. Performance → Why is it slow if there are no obvious errors? (10–30 minutes)
```

Most bugs are resolved by step 2 or 3. The Debugger and Performance panel are for problems that survive the earlier steps.

Part 1 — Visual and Layout Bugs

Symptom: element is missing, invisible, or in the wrong place

Open the **Inspector** panel (`F12` → Inspector, or right-click the element → Inspect Element).

Common causes and how to find them in the Inspector:

1. `display: none` **or** `visibility: hidden`

In the Inspector, select the element. In the Rules panel on the right, look for `display: none` or `visibility: hidden`. A strikethrough on any property means it was overridden by a more specific rule.

```
/* This rule is hiding the element */
.modal-overlay { display: none; }

/* A higher-specificity rule overriding it */
.modal-overlay.active { display: flex; }
```

If `display: none` appears without an `.active` override present, the JavaScript that should add the `active` class never ran — check the Console for errors.

2. Element is off-screen

In the Rules panel, look for large negative margins (`margin-left: -9999px`), `position: absolute` with extreme `top`/`left` values, or `overflow: hidden` on a parent clipping the element.

Select the element in the Inspector → it highlights in the page with a blue overlay. If you can't see the highlight, the element is likely scrolled off-screen or clipped. Check `overflow` on the parent.

3. `z-index` conflicts

An element may be present but covered by another. In the Rules panel, search for `z-index` on the element and its siblings/ancestors. A lower `z-index` means it's behind another element.

Use the Inspector's **3D View** button (the cube icon in the Inspector toolbar) to visualise the stacking order of all elements.

4. Wrong CSS rule winning

In the Rules panel, rules are listed in order of specificity — the topmost rule is the one that wins. Overridden rules are shown with a strikethrough. Click the filename:line-number next to any rule to jump to it in the Style Editor.

5. Media query mismatch

If the layout breaks at a specific viewport width, use the **Responsive Design Mode** (`Ctrl + Shift + M`) to resize the viewport and watch the Rules panel. Rules inside a `@media` block are greyed out when their condition is not met.

Symptom: image or font not loading

1. Open the **Network** panel and filter by **Images** (or **Fonts**).
2. Look for the resource in the request list — is it there at all?

• Not in the list → the URL was never requested. Check the `src` or `href` attribute in the Inspector.

• In the list with a 404 → the file doesn't exist at that path. Check for typos.

• In the list with a 200 → the file loaded. Check the `alt` text and whether `display: none` is hiding the `<img>` element.

Symptom: layout breaks at a specific screen width

1. Open Responsive Design Mode (`Ctrl + Shift + M`).
2. Set the viewport to the width where the layout breaks.

3. In the Inspector → Rules panel, watch which `@media` rules activate and deactivate as you resize.
4. Click the `@media` label in the Rules panel — it jumps to the CSS file at the exact breakpoint declaration.

Part 2 — JavaScript Errors

Symptom: something on the page doesn't work after a click

Open the **Console** panel. JavaScript errors appear in red. This is always the first step.

Reading a Console error:

```
Uncaught TypeError: Cannot read properties of undefined (reading 'email')
at UserProfile (app.js:142:18)
at renderDashboard (app.js:87:5)
at main (app.js:12:3)
```

This tells you:

- **Error type**: `TypeError` — you tried to read a property of `undefined`
- **What was undefined**: the object at `app.js:142:18` — the line that has `.email`
- **Call stack**: the error bubbled up through `renderDashboard`, which was called from `main`

Click `app.js:142` in the stack trace — Firefox opens that line directly in the Debugger.

Finding the bug without waiting for a crash

Sometimes the page doesn't crash — it just produces wrong output. Use **console.log** strategically:

```
// In the Console, intercept an object you want to inspect
const user = JSON.parse(localStorage.getItem('user'));
console.log('User:', user); // Is the object what you expect?
console.table([user]); // Column view — easier to scan
console.dir(user); // Full prototype chain
```

Or set a **Watchpoint** on a variable:

1. In the Debugger, pause execution (set a breakpoint on a nearby line).
2. In the Scopes panel, right-click a variable → **Break on change**.
3. Resume — the Debugger will pause automatically the next time that variable's value changes.

Common JavaScript bugs and their Console signatures

Console message	Likely cause
Cannot read properties of undefined (reading 'X')	An object you expected to exist is undefined — API returned an unexpected shape, or data hasn't loaded yet

Console message	Likely cause
X is not a function	You called something that isn't a function — likely a variable shadowing a function name
ReferenceError: X is not defined	Variable used before declaration, or a typo in the name
Uncaught (in promise) Error: …	An async function threw but there was no .catch() or try/catch
Failed to fetch	A fetch() call failed — check the Network panel for the corresponding request
SyntaxError: Unexpected token '<'	A fetch() expected JSON but got HTML (likely a 404 error page) — check the Network panel response body

Debugging an async / Promise error

When an error message says "Uncaught (in promise)", the error was thrown inside an async function with no error handler. In the Debugger:

1. Click the **Exception** breakpoint icon (■ hexagon) in the Debugger toolbar.
2. Select **Pause on uncaught exceptions**.
3. Trigger the failing action.
4. The Debugger pauses at the exact line that threw — no guessing.

Part 3 — Slow Page Loads

Triage in the Network panel

1. Open the Network panel.
2. Click the **Reload** button (■) in the toolbar — or close and reopen DevTools, then reload the page — to capture the full load.
3. Look at the **DOMContentLoaded** (blue vertical line) and **Load** (red vertical line) markers at the bottom of the request list. Note their times.

Sorting by Size:

Click the **Size** column header to sort requests by size, largest first. A 3 MB JavaScript bundle on the critical path is the single most common cause of slow page loads.

Sorting by Time:

Sort by the **Time** column. Look for requests that took more than 500ms. Click each one → **Timings** tab. Is the time dominated by Waiting (TTFB — server response time) or by Receiving (large payload)?

- Long **Waiting**: the server is slow to respond — database query, server computation, or cold start.
- Long **Receiving**: the payload is large — consider compression (gzip/Brotli) or pagination.

Render-blocking resources

Scripts and stylesheets that load in the `<head>` without `defer` or `async` block HTML parsing — nothing else can render until they finish downloading and executing. In the Network panel, look for:

- A JavaScript file early in the waterfall that takes a long time to load.
- Everything else starting only after that file finishes (a visible gap in the waterfall).

In the Inspector, check the `<head>` of the document:

```
<!-- Render-blocking — bad -->
<script src="/app.js"></script>

<!-- Non-blocking — good -->
<script src="/app.js" defer></script>
<script src="/analytics.js" async></script>
```

Third-party script impact

Third-party scripts (analytics, chat widgets, A/B testing tools) often load slowly and block the main thread. In the Network panel, click the **Domain** column header. Requests from domains other than your own are third-party. Sort by Time and note how long each third-party request takes.

Cache effectiveness

In the Network panel, check the **Transferred** column vs the **Size** column:

- If **Transferred** says `0B` (cached) → the resource came from the browser cache. ■
- If **Transferred** equals **Size** → the resource was downloaded fresh. Consider setting `Cache-Control` headers.

Reload with `Ctrl + Shift + R` (hard reload, clears cache) to see the cold-load time. Reload with `F5` to see the warm-load time. The difference is your caching benefit.

Part 4 — Runtime Performance (Jank and Freezes)

When the page loads fine but becomes slow or unresponsive during use, the **Performance panel** is the tool.

Recording the problem

1. Open the Performance panel (`Ctrl + Shift + P`).
2. Click **Record** (or `Ctrl + E`).
3. Perform the action that is slow — scroll the page, click a button, open a modal.
4. Stop immediately after the slowness occurs. Keep recordings short (3–5 seconds).
5. Look for red sections in the FPS bar — these are dropped frames.

Reading the flame chart

Zoom into a dropped-frame section (drag on the overview strip). Look for:

- **A wide yellow bar** — long JavaScript task. Click it → Details panel shows which function.

- **Repeated purple bars** — layout recalculations. Common cause: layout thrashing.
- **A red triangle** in the top-right corner of a bar — this is a **long task** (> 50ms).

Diagnosing layout thrashing

Layout thrashing occurs when code reads a layout property (like `offsetWidth`) after writing a style change, forcing the browser to recalculate layout synchronously. It appears in the flame chart as many small purple (Layout) bars inside a JavaScript task.

Identifying it in the flame chart:

1. Click the outer yellow (JS) bar.
2. In the Details panel → Call Tree, look for `recalculate styles` and `layout` entries nested repeatedly inside your function.
3. Click the Call Tree entry — it shows the source file and line number.

The pattern to find in your code:

```
// Bad — reads offsetWidth AFTER setting width, forcing layout recalc each time
elements.forEach(el => {
el.style.width = container.offsetWidth + 'px'; // read + write per loop
});

// Good — read once outside the loop
const w = container.offsetWidth;
elements.forEach(el => {
el.style.width = w + 'px';
});
```

Diagnosing an expensive event handler

If the page freezes on a specific user action (click, scroll, resize):

1. In the Performance panel, click **Record**, trigger the action, stop.
2. In the flame chart, find the event name (e.g. `click`, `scroll`) — click it.
3. The Details panel shows the call tree for that event handler.
4. Look for the function with the highest **Self Time** in the Bottom-Up tab.

Alternatively, use an **Event Listener Breakpoint** in the Debugger:

1. Open the Debugger panel.
2. In the right panel, expand **Event Listener Breakpoints**.
3. Expand **Mouse** → check **click**.
4. Click the element. The Debugger pauses at the first line of the click handler — step through to find the slow operation.

The Bottom-Up tab shortcut

After recording any performance profile, the fastest route to the bottleneck is:

1. Click anywhere in the flame chart that looks expensive.

2. Open the Details panel → **Bottom-Up** tab.
3. Sort by **Self Time** descending.
4. The top entry is the function doing the most work. Click it to jump to it in the flame chart.

Part 5 — Cross-Panel Workflow Example

Here is how a real diagnosis might unfold when a user reports: *"The dashboard loads but then freezes every time I click 'Load More'."*

Step 1 — Console (10 seconds)

Open DevTools, click Load More. Any red errors? If yes — fix the error first. If no — continue.

Step 2 — Network (1 minute)

Filter by XHR/Fetch. Click Load More. Watch the Network panel:

- Does a request fire? If not → the click handler isn't making the call. Check the Debugger.
- Does it fire but return an error status (500, 422)? → Server-side issue. Check the Response body.
- Does it fire and return 200 but the page still freezes? → The JavaScript processing the response is slow. Go to Performance.

Step 3 — Performance (5 minutes)

Record while clicking Load More. Find the click event in the flame chart. Bottom-Up → Self Time. If the top entry is `JSON.parse` or a sorting/filtering function, that's the bottleneck — the response is too large or the processing is unoptimised.

Step 4 — Debugger (if needed)

Set a breakpoint at the start of the `loadMore` function. Step through it. Watch the Scopes panel — at what point does the data look wrong or unexpectedly large?

Practical Tips

`Ctrl + Shift + R` vs `F5`

`F5` is a normal reload (uses cache). `Ctrl + Shift + R` is a hard reload (ignores cache). Always use `Ctrl + Shift + R` when investigating resource loading — otherwise you might be looking at a cached version of a file you already fixed.

The Console `$0` shortcut

After selecting an element in the Inspector, switch to the Console and type `$0`. This gives you a reference to that element in JavaScript — you can call methods on it, read its properties, and check its event listeners without writing any setup code.

Check event listeners from the Inspector

In the Inspector, select any element. In the right panel, look for the **Event Listeners** tab (next to Rules, Computed, etc.). This lists every JavaScript event handler registered on that element — including the

function name, file, and line number. Click any handler to jump to it in the Debugger. This is the fastest way to find "what runs when I click this button."

Network: disable cache while DevTools is open

In the Network panel toolbar, check **Disable Cache**. This forces every resource to download fresh on every reload — so you always see the real network time, not a cached result. Remember to uncheck it when you're done.

Use the Console as a REPL to reproduce bugs

You don't always need to click through the UI to reproduce a bug. Call the relevant function directly from the Console: `window.loadMore(1, 20)`. This isolates the function from the UI event handler and makes it much faster to iterate on a fix.

Exercises

1. Open any page with a form or button. Right-click the button → Inspect Element. In the Inspector's right panel, click the **Event Listeners** tab. What function is registered for the `click` event? What file and line is it on?

2. Open the Console on any page and run:

```
document.querySelectorAll('img').forEach((img, i) => {
if (!img.complete || img.naturalHeight === 0) {
console.warn(`Image ${i} failed to load:`, img.src);
}
});
```

Are any images broken? If so, switch to the Network panel → filter by Images to find the failing request.

3. Open the Network panel and reload a page. Sort by **Size** descending. What is the largest resource? What type is it (JS, CSS, image)? Is it compressed (check the Size vs Transferred columns)?

4. In the Network panel, sort by **Time** descending. Click the slowest request → **Timings** tab. Is the time dominated by Waiting (TTFB) or Receiving? What does that tell you about where the slowness is?

5. Open the Performance panel (`Ctrl + Shift + P`). Click **Record**, scroll the page for 3 seconds, stop. Are there any red sections in the FPS bar? Click a dropped-frame section and open the Bottom-Up tab. What function has the highest Self Time?

6. In the Console, type:

```
performance.getEntriesByType('navigation')[0]
```

Read the `domContentLoadedEventEnd`, `loadEventEnd`, and `transferSize` values. What do these tell you about the page load?

7. Enable **Disable Cache** in the Network panel toolbar. Hard-reload the page (`Ctrl + Shift + R`). Record the Load time. Now uncheck Disable Cache and reload again. What is the difference in Load time between cached and uncached?

Key Takeaways

- **Triage in order**: Console → Network → Inspector → Debugger → Performance. Most bugs are found before step 4.
- **Console errors include a stack trace** — click the file:line link to jump directly to the problem in the Debugger.
- **"SyntaxError: Unexpected token '<'"** after `fetch()` means the server returned HTML (probably an error page) instead of JSON — check the Network panel Response body.
- **Sort Network by Size** to find large resources; **sort by Time** to find slow ones. Check Timings to tell TTFB from payload size.
- **Render-blocking scripts** without `defer` or `async` delay everything that loads after them — visible as a gap in the waterfall.
- **Layout thrashing** (read → write → read → write in a loop) shows as repeated purple bars inside a JS task in the Performance flame chart.
- **Bottom-Up** → **Self Time** is the fastest route to the most expensive function in any performance recording.
- **Event Listeners tab** in the Inspector is the fastest way to find what JavaScript runs when you interact with an element.
- `$0` in the Console gives you a live reference to the element selected in the Inspector.
- **Disable Cache** in the Network panel toolbar ensures you're always seeing real network timings, not cache hits.

What's Next

Lesson 11 is the final scenario — **Reverse-Engineering an API** — using the Network panel, Console, and Debugger to discover, document, and replay an undocumented API that a web application uses internally.

LESSON 11 OF 11 · SCENARIO

Scenario: Reverse-Engineering an API

Overview

This is the final lesson. The technique taught here — using DevTools to discover and replay the API calls a web application makes internally — is one of the most practical skills a developer can have. You use it when:

- Building an integration with a service that has no public API documentation.
- Checking whether a third-party app communicates with endpoints you didn't expect.
- Debugging a mobile or web app by inspecting exactly what it sends and receives.
- Replicating a request in a test, a script, or Postman to automate something the UI does manually.

By the end of this lesson you will be able to:

- Intercept and read every API call a web application makes.
- Understand the request structure: URL patterns, headers, authentication, and request bodies.
- Replay any request from the Console, a terminal, or a script.
- Build a working API map from a series of observed requests.
- Recognise common pagination, filtering, and versioning patterns.

A Note on Ethics and Legality

Reverse-engineering an API for your **own use** — building a personal script, debugging your own application, or understanding what an app you own does — is generally legal and entirely normal engineering work.

Reverse-engineering an API to **circumvent rate limits, scrape data at scale, violate a service's Terms of Service, or access data you are not authorised to view** is a different matter. Many jurisdictions have laws (the CFAA in the US, the Computer Misuse Act in the UK, GDPR in the EU) that may apply depending on what you do with the information you find.

The techniques in this lesson are neutral tools. Use them responsibly.

Step 1 — Set Up the Network Panel

Before interacting with the application, prepare the Network panel to capture everything cleanly.

1. Open DevTools (`F12`) → **Network** panel.
2. Check **Disable Cache** in the toolbar — forces fresh requests every time, so you see the real traffic.
3. Check **Persist Logs** (right-click the toolbar → Persist Logs, or look for the checkbox) — keeps the request list intact across page navigations and redirects.

4. Clear any existing requests with the ■■ trash icon.
5. Set the filter to **XHR/Fetch** — this hides HTML, CSS, images, and fonts, leaving only the API calls you care about.

You are now ready to observe.

Step 2 — Trigger Actions and Observe Requests

Navigate through the application, performing the actions you want to understand. Each action (clicking a button, submitting a form, scrolling, filtering) likely triggers one or more API calls. Watch the Network panel as you act.

Finding the right request

In a busy application, dozens of requests may fire at once. Narrow them down:

By URL text filter:

Type a keyword in the Network panel's filter box. If you're clicking a "Load users" button, type `users` or `api` in the filter.

By timing:

Click the **Waterfall** column header to sort by start time. The request that fired immediately after your click is the one you want.

By method:

Filter to see only `POST` requests when you submit a form, or only `GET` when you load a list.

By searching inside responses:

Press `Ctrl + F` in the Network panel. Type a value you know appeared in the UI (a username, a price, a product ID). The search scans all response bodies and highlights the request that returned it.

Step 3 — Read and Document the Request

Click the request in the Network panel. The detail pane opens on the right (or below). Work through each tab:

Headers tab

This is the most important starting point. Record:

Request URL — the full endpoint:

```
https://api.example.com/v2/products?category=electronics&page=1&limit=20
```

Break it into parts:

- **Base URL**: `https://api.example.com`

- **Version prefix**: `/v2/`
- **Resource path**: `products`
- **Query parameters**: `category=electronics`, `page=1`, `limit=20`

Request Method: `GET`, `POST`, `PUT`, `PATCH`, `DELETE` — this tells you what operation the endpoint performs.

Request Headers — look for auth and content type:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Content-Type: application/json
Accept: application/json
X-Client-Version: 3.2.1
X-Request-ID: 7f4e2d1a-9b3c
```

Write down every non-standard header (those starting with `X-`). They often carry API version, client ID, or signature values that the server validates. Replaying a request without these headers may return a different response or a 400/403 error.

Response Headers — look for rate limits and versioning:

```
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 987
X-RateLimit-Reset: 1730000000
Content-Encoding: gzip
Cache-Control: max-age=300, private
```

Rate-limit headers tell you how many requests you can make per window. If your script hits the limit, it will get 429 responses.

Request tab (for POST/PUT/PATCH)

Click the **Request** tab to see the body sent with write operations. Common formats:

```
// JSON body (most modern APIs)
{
"name": "New Product",
"price": 29.99,
"category_id": 7,
"tags": ["electronics", "sale"]
}
```

```
// Form-encoded body (older APIs and login endpoints)
email=alice%40example.com&password=••••••••&remember=true
```

Response tab

Click the **Response** tab. This shows exactly what the server returned. Note:

- The top-level structure (object vs array vs wrapped object like `{ "data": [...] }`)
- The field names and data types
- Which fields are IDs you can use in subsequent requests

- Any metadata fields (pagination, totals, cursors)

Example response with pagination:

```
{
"data": [
{ "id": 1, "name": "Widget A", "price": 9.99 },
{ "id": 2, "name": "Widget B", "price": 14.99 }
],
"meta": {
"total": 847,
"page": 1,
"per_page": 20,
"last_page": 43
}
}
```

From this single response you know:

- There are 847 total items.
- Pagination uses `page` and `per_page` parameters.
- To get the second page, change `page=1` to `page=2`.
- You need 43 requests to get all items.

Step 4 — Replay the Request

Once you understand the request, replay it to verify your understanding and start experimenting.

Method 1 — Copy as cURL

Right-click any request in the Network panel → **Copy as cURL**. This copies a complete `curl` command with all headers, cookies, and the request body:

```
curl 'https://api.example.com/v2/products?category=electronics&page=1&limit=20' \
-H 'Authorization: Bearer eyJhbGci...' \
-H 'Content-Type: application/json' \
-H 'X-Client-Version: 3.2.1' \
--compressed
```

Paste this into a terminal. It should return the same response as the browser received.

Once the curl works, you can modify parameters:

```
# Get page 2
curl 'https://api.example.com/v2/products?category=electronics&page=2&limit=20' ...

# Increase the page size
curl 'https://api.example.com/v2/products?category=electronics&page=1&limit=100' ...
```

Method 2 — Replay from the Console

The Console lets you replay and modify requests in JavaScript without leaving the browser. The browser automatically includes the current session's cookies, so you don't need to copy the auth headers manually for session-cookie-authenticated APIs.

```
// Replay a GET request
const response = await fetch('https://api.example.com/v2/products?page=2&limit=20');
const data = await response.json();
console.table(data.data);
```

For Bearer token APIs, add the Authorization header:

```
const token = localStorage.getItem('access_token');
const response = await fetch('https://api.example.com/v2/products', {
headers: {
'Authorization': `Bearer ${token}`,
'Content-Type': 'application/json'
}
});
const data = await response.json();
console.table(data.data);
```

For POST requests:

```
const token = localStorage.getItem('access_token');
const response = await fetch('https://api.example.com/v2/products', {
method: 'POST',
headers: {
'Authorization': `Bearer ${token}`,
'Content-Type': 'application/json'
},
body: JSON.stringify({
name: 'Test Product',
price: 9.99,
category_id: 7
})
});
const result = await response.json();
console.log('Created:', result);
```

Method 3 — Edit and Resend

Firefox has a built-in request editor. Right-click any request in the Network panel → **Edit and Resend**. A panel opens where you can:

- Change the URL (add, remove, or modify query parameters)
- Change the method
- Edit the request headers
- Edit the request body

Click **Send** — the modified request fires and appears in the Network panel list like any other request, with a full response you can inspect.

This is the fastest way to test small variations without writing any code.

Step 5 — Identify Patterns Across Multiple Requests

A single request tells you about one endpoint. Trigger several different actions and look for patterns across the requests.

URL naming conventions

Watch for consistent patterns in how endpoints are named:

Pattern	What it usually means
/api/v1/users	REST resource list (GET = list, POST = create)
/api/v1/users/42	REST resource by ID (GET = read, PUT = update, DELETE = delete)
/api/v1/users/42/orders	Nested resource — orders belonging to user 42
/graphql	Single endpoint — all operations in the POST body
/rpc/getUsers	RPC style — operation name in the URL
?action=listProducts	Old-style RPC hidden in query params

Pagination patterns

Field name	Pattern type
page + per_page / limit	Page-based — predictable, easy to iterate
offset + limit	Offset-based — same as page-based but in absolute terms
cursor or next_cursor	Cursor-based — for large datasets; copy the cursor from one response into the next request
next_page_token	Token-based — similar to cursor
after (in GraphQL)	Relay-style cursor

Authentication patterns

Observe the `Authorization` header (or lack thereof) across requests:

- Same Bearer token on every request → stateless JWT auth
- `Cookie` header present, no `Authorization` → session cookie auth
- `X-API-Key` header → API key auth, usually static
- `Authorization` changes between requests → short-lived tokens with refresh, or per-request signing

Finding hidden parameters

Try modifying query parameters on an observed GET request to discover undocumented filtering or sorting options:

```
# Observed request
/api/products?page=1&limit=20

# Try adding likely parameters — some may work
/api/products?page=1&limit=20&sort=price_asc
/api/products?page=1&limit=20&search=widget
/api/products?page=1&limit=20&category=electronics&in_stock=true
/api/products?page=1&limit=20&fields=id,name,price
```

If a parameter is supported but not documented, the server will use it. If not supported, it will be silently ignored (or occasionally return a 400).

Step 6 — Find JavaScript Source Clues in the Debugger

The Network panel shows you what the application sends. The Debugger shows you *why* — the code that constructs the requests. This is useful when you want to understand all possible parameter values, not just the ones triggered by clicking through the UI.

Finding the fetch call

1. In the Network panel, click the request you want to understand.
2. Click the **Initiator** column value (or look for "Initiator" in the request detail pane). This shows the JavaScript file and line that called `fetch()` or `XMLHttpRequest`.
3. Click the file:line link — Firefox opens the Debugger at the exact line that made the request.

Reading the call site

At the call site you can see:

- All parameters being constructed — including ones the UI never exposes.
- Conditional logic that adds certain headers only in specific states.
- How the token is retrieved and formatted.
- Any request signing or HMAC calculation.

```
// What you might find at the call site
async function loadProducts(filters) {
const params = new URLSearchParams({
page: filters.page || 1,
limit: filters.pageSize || 20,
category: filters.category,
sort: filters.sortField + '_' + filters.sortOrder, // e.g. "price_asc"
in_stock: filters.showInStockOnly ? 'true' : undefined
});

return fetch(`/api/v2/products?${params}`, {
headers: {
'Authorization': `Bearer ${getToken()}`,
'X-Client-Version': APP_VERSION
}
});
}
```

From reading this function you now know that `sort` takes the format `field_direction` (e.g. `price_asc`, `name_desc`) — information that never appears in the UI but is fully documented in the source.

Setting a breakpoint at the call site

1. Open the Debugger at the call site.
2. Click the line number of the `fetch()` call to set a breakpoint (blue dot ●).
3. Trigger the action in the UI.
4. The Debugger pauses before the request fires.
5. In the Scopes panel, inspect the `params` variable — you can see every parameter that will be sent, including ones derived from state you can't see in the URL.

Building an API Map

As you work through the application, document your findings. A simple API map captures everything you need to replay the API programmatically:

```
BASE URL: https://api.example.com
AUTH: Bearer token — from POST /auth/login, stored in localStorage ('access_token')
Token expires in 3600s — check exp claim. Refresh: POST /auth/refresh

ENDPOINTS:

GET /v2/products
Query params: page (int), limit (int, max 100), category (string), sort (field_direction),
in_stock (bool)
Response: { data: Product[], meta: { total, page, per_page, last_page } }

GET /v2/products/:id
Response: { data: Product }

POST /v2/products
Body: { name (string, required), price (float, required), category_id (int), tags
(string[]) }
Response: { data: Product }

GET /v2/categories
No params
Response: { data: Category[] }

Rate limits: X-RateLimit-Limit: 1000/hour per token
X-RateLimit-Remaining: in every response header
429 when exceeded — retry after X-RateLimit-Reset (Unix timestamp)
```

This map is the deliverable. With it, you can write scripts, build integrations, or generate OpenAPI documentation — all from observing a browser session.

Practical Tips

Search response bodies to find which endpoint returns a specific value

In the Network panel, press `Ctrl + F` and type a value you see in the UI (a user's name, a price, an ID). The search scans all response bodies — the highlighted result is the request that returned that data.

Use `console.table()` to explore large API responses

After replaying a request in the Console, `console.table(data.data)` renders the response array as a sortable table — far easier to scan than a collapsed JSON tree.

Check the Initiator column before using the Debugger

The Initiator column in the Network panel often names the function that made the call. If the name matches something recognisable in the source, you may not need to set a breakpoint — just search for it with `Ctrl + P` in the Debugger.

Compare a working and a failing request with HAR

When a request works in the browser but your script's version fails, save a HAR file from the Network panel (right-click → Save All as HAR) and open it in a HAR viewer. Compare your script's request against the browser's request header by header. The difference is the missing piece.

Rate limit awareness

If you're writing a script that calls the API repeatedly, read the `X-RateLimit-Remaining` header from each response. When it approaches zero, sleep until `X-RateLimit-Reset` (a Unix timestamp). Hitting the limit gets your token blocked.

Exercises

1. Open the Network panel on any web app you use (a shopping site, a news site, your email provider). Filter by XHR/Fetch. Click through the UI and find a GET request that returns a list of items. Click the request → Response tab. What is the top-level structure of the JSON?

2. Find a request with a query parameter like `page=1` or `offset=0`. Try replaying it in the Console with `page=2`. Does the response contain different items?

3. Right-click any XHR/Fetch request → **Copy as cURL**. Paste it into a terminal. Does it return the same response? Now modify one query parameter and run it again.

4. Find a request in the Network panel. Click its **Initiator** value. Does it open the Debugger at the `fetch()` call? What function is making the request?

5. In the Console, replay an API call you observed and use `console.table()` to display the response data:

```
const r = await fetch('URL_FROM_NETWORK_PANEL');
const d = await r.json();
console.table(d.data || d); // adjust path to the array in the response
```

6. Find a paginated endpoint (one with `page` or `offset` in the URL). Write a small Console snippet that fetches pages 1 through 3 and combines the results:

```
const pages = await Promise.all(
[1, 2, 3].map(p =>
fetch(`/api/items?page=${p}&limit=20`)
.then(r => r.json())
.then(d => d.data)
)
);
const allItems = pages.flat();
console.log('Total items fetched:', allItems.length);
console.table(allItems);
```

7. Open the Network panel and look for any request that has `X-RateLimit-Remaining` in its response headers. What is the limit? How many requests have you used?

Key Takeaways

- **Filter to XHR/Fetch** first — this removes all the noise (HTML, CSS, images) and leaves only API calls.

- **Ctrl + F in the Network panel** searches inside response bodies — use it to find which request returned a specific value you see in the UI.
- **Every request detail is in four tabs**: Headers (URL, method, auth), Request (body for write operations), Response (the data returned), Timings (performance breakdown).
- **Copy as cURL** (right-click any request) gives you an instantly replayable command with all headers — the fastest way to get a request out of the browser and into a script.
- **Replay from the Console** with `fetch()` — the browser automatically includes the current session's cookies, so session-authenticated requests work without copying auth headers.
- **Edit and Resend** (right-click → Edit and Resend) lets you modify and re-fire any request without writing code.
- **The Initiator column** links directly to the JavaScript that made the request — click it to find the call site in the Debugger.
- **Pagination is discoverable** from the response metadata — look for `total`, `page`, `per_page`, `cursor`, or `next_page_token` fields.
- **Rate limit headers** (`X-RateLimit-Remaining`, `X-RateLimit-Reset`) are in the response headers — read them before writing scripts that make many requests.
- **Use responsibly** — these techniques are normal engineering tools. Automating requests against a service you don't own, at scale, or in violation of their ToS may have legal consequences.

Course Complete

You have now covered all 11 lessons in this Firefox DevTools course:

Lesson	Topic
1	Introduction to DevTools
2	The Inspector Panel
3	The Console Panel
4	The Network Panel (with HAR files)
5	The Debugger Panel
6	The Storage Panel
7	The Performance Panel
8	Scenario: Inspecting Authentication
9	Scenario: Troubleshooting Client Access
10	Scenario: Debugging a Slow or Broken Page
11	Scenario: Reverse-Engineering an API

The scenario lessons (8–11) are designed to be revisited whenever you face those real-world situations. The tool-specific lessons (2–7) are reference material — return to them when you need a reminder of what a specific panel can do.