8

Debugging and profiling

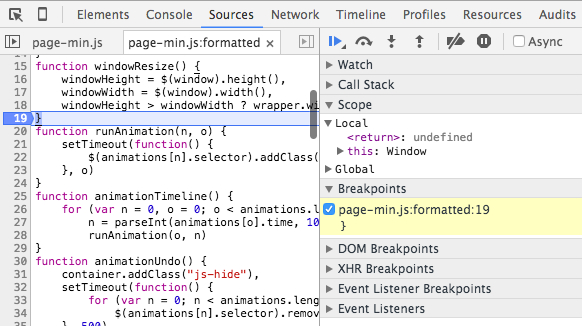
Debugging is a tricky part of programming. Bugs during development are unavoidable. Whatever our experience we have to spend quite a time on hunting them. It happens by the look of the code there must be no problem with the application yet the developer fights it for hours until running into a silly reason like misprinted property name. Most of this time could be saved by making better use of browser development tools. So we will consider in this chapter the following topics:

* How to discover bugs
* Getting the best from console API
* How to tune performance

# Hunting bugs

Debugging is about finding and resolving defects that prevent the intended application behavior. Where crucial is to find the code causing the problem. What do we usually do when encountering a bug? Let’s say, we have a form that is assumed to run validation on submit event, but it doesn’t. First of all we have a number of assumption to validate. For example, if the reference to the form element is valid; if the event and method name spelled correctly during registering a listener; if the object context is not lost in the body of the listener, and so on.

Some bugs can be discovered automatically like by validating input and output on entry and exit points of methods (Design by Contract https://en.wikipedia.org/wiki/Design\_by\_contract). But other bugs we have to spot manually and here can be two options. Going from the point where the code is surely correct step by step to the problem point (Bottom-Up debugging), or on the contrary stepping back from the break point to find the source of the break. Here browser development tools became handy. The most advanced one is Chrome DevTools. We can open Source Panel in it and set breakpoints in the code. The browser stops execution while reaching a breakpoint and shows a pane with actual variable scope and call stack. It also provides controls that one can use to “step-through” the code back and forth one line at a time.



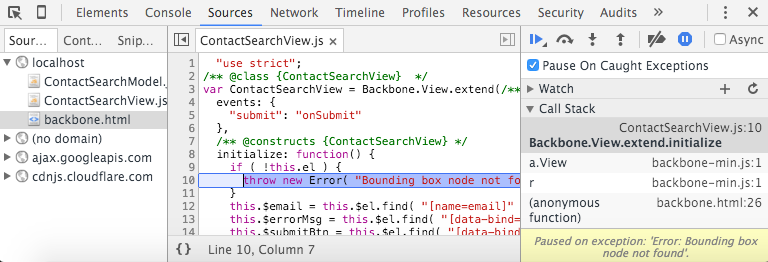
Insert image B04886\_08\_01.png

However that can be tricky to navigate through the codebase in DevTools. Fortunately you can set a break-point out of the browser, directly in the IDE. You just need to put debugger; statement on the line where you want the browser to break.

Sometimes it is hard to figure out what’s going on with the DOM. We can make DevTools to break on DOM events such as node removal, node modification and subtree changes. Just navigate to the HTML element in the Source Panel, right-click and choose “Break on…” option.

Besides in the Source Panel there is tab called XHR Breakpoints where we can set a list of URLs. The browser will break then when any of the URLs is requested.

You can also find in the top of Source Panel sidebar an icon in form stop sign. If pressing this button down DevTools will break on any caught exception bringing you to the throw location in the source code.



Insert image B04886\_08-02-07.png

For more information, consults with https://developer.chrome.com/devtools/docs/javascript-debugging

# Getting the best from console API

Despite it’s no part of JavaScript we all use Console API extensively to find out what is really happening during the app life-cycle. The API once introduced by Firebug tool now is available in every major JavaScript agent. Most developers do just simple logging using methods error, trace, log and the decorators info and warn. Well, when we pass any values into console.log they get presented in the JavaScript Console panel. Usually we pass a string describing the case and a list various objects that we want to inspect. But did you know that we can refer these object directly from the string in the manner PHP sprintf? So the string given as the first argument can be a template that contains format specifiers for the rest of arguments:

var node = document.body;

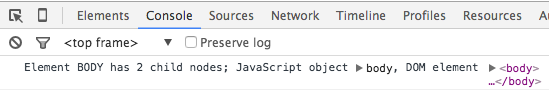
console.log( "Element %s has %d child nodes; JavaScript object %O, DOM element %o",

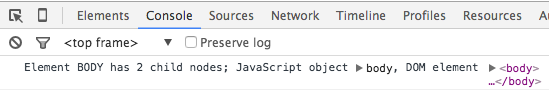
node.tagName,

node.childNodes.length,

node,

node );



Insert image B04886\_08\_02.png

The available specifiers are %s for strings, %d for numbers, %o for DOM elements, %O for JavaScript objects (the same as console.dir). Besides, there is a particular specifier that allows us to style console.log report. That can be very useful. In practice the application console receives too many log records. It gets hard to make out desired messages among hundredth alike. What we can do is to categorize the messages and style them accordingly.

console.log.user = function(){

var args = [].slice.call( arguments );

args.splice( 0, 0, "%c USER ",

"background-color: #7DB4B5; border-radius: 3px; color: #fff; font-weight: bold; " );

console.log.apply( console, args );

};

console.log.event = function(){

var args = [].slice.call( arguments );

args.splice( 0, 0, "%c EVENT ",

"background-color: #f72; border-radius: 3px; color: #fff; font-weight: bold; " );

console.log.apply( console, args );

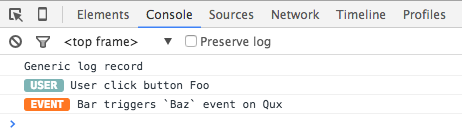
};

console.log( "Generic log record" );

console.log.user( "User click button Foo" );

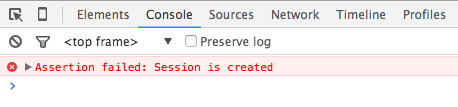
console.log.event( "Bar triggers `Baz` event on Qux" );

In this example we define two methods extending console.log. One prefixes console messages with USERon cyan and is intended for user action events. The second prepends report with EVENT and is meant to highlight mediator events.

Insert image B04886\_08\_03.png

Another less-known trick is to use console.assert for assertions in the code logic. So we assume that a condition is true and until it so - everything is fine and we get no messages. But as soon as it failed, we get a record in the console

console.assert( sessionId > 0, "Session is created" );

Insert image B04886\_08\_04.png

Sometimes we need to know how often an event happens. Here we can use method console.count:

function factory( constr ){

console.count( "Factory is called for " + constr );

// return new window[ constr ]();

}

factory( "Foo" );

factory( "Bar" );

factory( "Foo" );

It displays in the console the specified message and auto-updating counter next to it.

Insert image B04886\_08\_05.png

You can find more on working with console at https://developer.chrome.com/devtools/docs/console

# Tuning performance

Performance is user experience. If it takes too long to load a page or UI to respond the user is likely to leave the application and never come back. It’s especially true with web apps. In *chapter 3 “DOM Scripting and AJAX”* we compared different approaches to manipulate the DOM. In order to find out how fast an approach was we used performance built-in object:

"use strict";

var cpuExpensiveOperation = function(){

var i = 100000;

while( --i ) {

document.body.appendChild( document.createElement( "div" ) );

}

},

// Start test time

s = performance.now();

cpuExpensiveOperation();

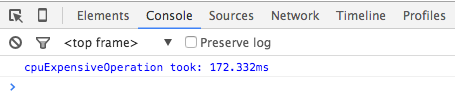
console.log( "Process took", performance.now() - s, "ms" );

performance.now() returns a high resolution timestamp that represents time in milliseconds accurate to microseconds. That is designed and widely used for benchmarking. However console object also provides methods to measure time time/timeEnd:

console.time( "cpuExpensiveOperation took" );

cpuExpensiveOperation();

console.timeEnd( "cpuExpensiveOperation took" );

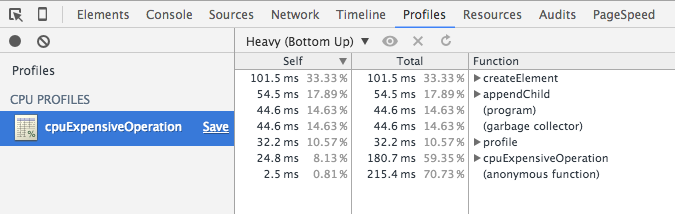
Insert image B04886\_08\_06.png

If we need to know what exactly going on during the operation execution we can request profile for that period:

console.profile( "cpuExpensiveOperation" );

cpuExpensiveOperation();

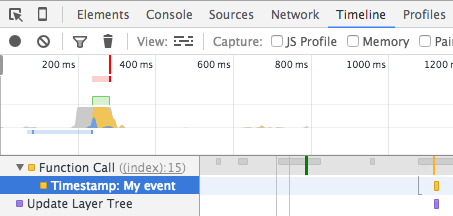
console.profileEnd( "cpuExpensiveOperation" );

Insert image B04886\_08\_07.png

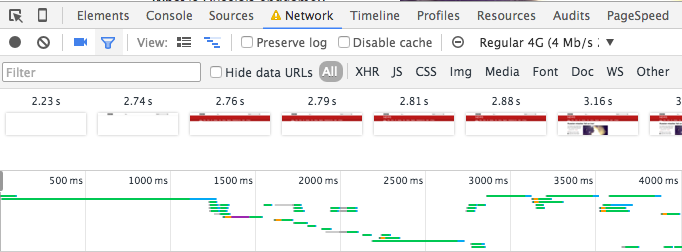
Moreover, we can mark the exact time of the event in the Timeline panel of DevTools:

cpuExpensiveOperation();

console.timeStamp( "cpuExpensiveOperation finished" );

Insert image B04886\_08\_08.png

When tuning performance we have to pay particular attention to the response time. There are a number of techniques that can be used to improve user experience during the bootstrap (non-blocking JavaScript and CSS loading, critical CSS, moving static files CDN and others). Well, let’s say you decide to load CSS asynchronously (https://www.npmjs.com/package/asynccss) and cache into localStorage. But how would you test if you gained anything by this? Fortunately DevTools has Filmstrip feature. We just need to open Network panel, enable Screenshot capturing and reload the page. DevTools shows us the progress of the page load, frame after frame, as the user sees the page during the load process. Besides, we we can manually set connection speed (throttling) for a test and find out how it affects the filmstrip.



Insert image B04886\_08-02-05.png

Summary

Debugging is an integral part of web development. It can be also a pretty sluggish and tedious task. With browser development tools we reduce the time spent on hunting the bugs. We set breakpoints in the code and move step by step to the source of the problem the same way as the program does. When using Chrome DevTools we can watch for DOM modification events and for specified URL requests. When tuning performance, we can measure time with time/timeEnd and request process profile with profile/profileEnd. By using features such Filmstrip and Throttling we can see looks the page load on different connections.

We started this book with reviewing JavaScript core features. We've learnt how to make the code more expressive by means of syntactic sugar, we practiced object iteration and collection normalization, we compared various approaches to declare object including ES6 classes, we found out how to use “magic methods” of JavaScript. Then we dove into modular programming. We talked about module pattern and modules in general and reviewed three main approaches to modularization in JavaScript AMD, CommonJS and ES6 modules. The next topic was about keeping DOM manipulations high-performance. We also examined Fetch API. We also considered some of most exciting HTML5 APIs such Storage, IndexedDB, Workers, SSE, WebSocket and the technologies are the hood Web Component. We considered techniques to leverage JavaScript event loop and to build no-blocking applications. We practices with design patterns in JavaScript and talked about concern separations. We wrote a simple application in three frameworks Backbone, Angular and React. We have tried out Node.js by creating a command-line utility and exposing a web-server. We also created a demo desktop application with NW.js and it's mobile varsion with PhoneGap. At last, we talked about bug hunting.