**Javascript Debugging**

Debugging is the process of finding and resolving Defects or Problems within the program that prevent the correct operation of Computer Software or a system.

Some tactics: Interactive Debugging, Control Flow Analysis, Unit Testing, Integration Testing, Log File Analysis, Monitoring at the Application or System Level, Memory Dumps and Profiling.

Interactive: Write, run hit problems. This is our approach. Using Chrome Developer Tools. We debug as it runs through the browser (Console Output Analysis then Low-Level Analysis with tools).

**JavaScript Debugging Tools**

* Unit Integration Test 🡪 Provides the initial feedback for new changes to the codebase. Test Results are the first line of defense against Bugs.
* Console Logging 🡪 Can provide Instant Information depending on how robust we create the Log Statements.
* JS Errors 🡪 Using Try-Catch and creating custom Error Messages that provides Feedback.
* Chrom Dev Tools 🡪 Debugging Tool: Provides capability to stop the code at any point in the process, and absorbe the state of the app, so the cause of issues can be determined.

**Chrome Dev Tools:**

Default on the desktop! Because of the Dev Tools are the stronger around.

**Open 🡪** F12, Ctrl+Shift+I, Right Click -> Inspect, Command+Option+I

**Select Element 🡪** Ctrl+Shift+C

* Style Tab: Shows the CSS of the selected Element, we can Edit it.
* Sources Panel: Where we will be Debugging JS

**Breakpoint 🡪** En F12 / Sources… Seleccionamos el File a utilizar y al hacer click en cualquier línea activamos el Breakpoint.

La ejecución del programa se pausará en dicha línea y aparecerá un cartel: “Paused in Debugger”, además de mostrar la Línea seleccionada.

Hay botón de Play (resume execution) que cotinuará hasta el próximo Breakpoint o el final.

Al lado del botón de Play está el botón Step Over que continuará a la siguiente Línea después del checkpoint, una por una y pasuando.

Hay 2 Flechas (arriba y abajo) que saltan entre las funciones declaradas en el code.

A la derecha hay un Option: Breakpoints, nos mostrará los breakpoints creados. Una sesión de Debug podrá tener cientos de Breakpoints.

**CallStack 🡪** Ponemos un Breakpoint en la last line de UpdateTrollBox(). Y vemos la Option Call Stack.

Vemos la Current Function that we are paused in and the Event that triggered it.

Everything CALLED up to this moment of the state, to easily understand the codes execution flow.

If we Hover over the lines of the code we will see the Prototype Values of each element.

**Logging 🡪** Creative use of Console.Log() can provide significant support for debugging.

* With Console.group(“name”) we cna group different lines of code into a single one:

Console.Group(“name”)

Log()

Log()

Console.GroupEnd()

* Console.Info(“Info Text”): We can log info as a informative Text.
* Console.Clear(): Will clear all the console up to this line. En las herramientas de Dev Tools(Consolse section) we can actviate: Preserve Log to revert the Clear function.

\*When debugging feel free to add more and more Console Logs\*

**Source Maps 🡪** Useful when using Libraries.

Links the Pre-Processed code to the original source code in the debugger. (to the UnBuilt state).

Fundamental cuando el code fue pasado de ES6/7/8 a ES5.

Cuando la App fue Transpolated (ES6 to ES5) por Babel o Minified by a pre-processor.

**Uglify:** (app.min.js) Shrinks the code down by eliminating White space, Renaming variables and functions to minimal charácter names, etc.

JQuery can request a min.js but when displaying it is an inteligible code.

Source Maps can be generated by the Pre Processor to load the original source files, meanwahile Chrome runs the minified versión.

Enable SourceMaps on the Dev Tools. (done)

**Console 🡪**  We can use the Console to define JS code. Podemos usar Shift+Enter para crear multile line code.

* Snippets: F12/Sources/Snippets. This will allow us to run code blocks or commands repetitevly as needed.

Right click on the page of Snippets to save the created Snippet.

Console Drawer: Will display the console as we create the snippets or test them.

* We can Access and Change values in the console.

Ej: Create a Breakpoint for a function and then call its name on the console.

Then we can change the value created by just calling the name of function + = “new value” or “changed”

Setp over (arrow symbol) to the Next Function Call and we can see the value is now Changed to the value we set before.

**Debugging**

**Ejemplo:**

**Date out of desired Format**

1. Ver el Code a simple vista a ver si encontramos el Bug.
2. Ver Console a ver si hay alguna ayudita.
3. Ir al Source Tab.

Sabemos que el problema es antes de Click Submit. Entonces vamos a la tab Event Listeners Breakpoints y selectionamos Mouse-> Click

Hacemos click en el Btn y vemos que Date se encarga de nuestra fecha.

1. Creamos un Watch.

Vemos que TrollPost.date debería contener la info: Not Available.

Vamos Stepping ahead paso a paso y vemos que cuando nos da el Date, no es el formato que queremos. Vemos que antes de dar el format esta COMENTADO el COMANDO PARA FORMATEAR LA DATE.

1. Arreglamos el BUG y right click Save. Probamos correr el code saltando los breakpoints y vemos el nuevo formato dado a Date.
2. Arreglamos en VSCode y reload el browser. Cancelamos los breakpoints haciendo click en el icono.

**Bug Solved Using:**

Event-Driven Breakpoint

Deactivate Breakpoints Control

Apply code fixes in the Source Tab and run them

Use of Watch Feature to track different Variables and Properties within our Code.

**More types of Breakpoints**

**Conditional Line-of-code 🡪** When we know the exact región of code that we need to investigate. Only pausing when a condition is True.

Right click en la line + Create Conditional.

Ej: trollPost.troll === “ ”.

-> Frenará si no hay values.

->Continuará sin break si tiene algún value

**DOM Change 🡪** We can use this to break when the Node or Elements selected are changed in the DOM.

Ej: Break when the values are about to be displayed in the DOM. Vamos a Elements Tab, DIV selectionado, Right Click /Break on/Subtree Modifications.

Vemos entonces que se Frena justo donde hay un “innerHTML” asociado a un value. Además a la derecha vemos que hay un nuevo DOM Breakpoint y una Scope data

**Function 🡪** Call these in the console. We use Debug (Function Name) so we can stop the code when this function is Called.

Ej: Stop when UpdateTrollBox() is called.

Ponemos Breakpoint normalmente en UpdateTrollBox.

En Console: debug(UpdateTrollbox) y vemos que devuelve el valor Undefined.

**Dev Tools Console**

**Options and Settings 🡪** Dev Tools console configurations. Adjust Tab Size, ESC: Console Drawer, Preserve Log, Show TimeStamps, Save Console Log.

**Write/Run Javascript 🡪**

**$()** Returns the first element that matches in CSS -> document.QuerySelector()

**$$()** Returns Array of all the Matches CSS -> document.QuerySelectorAll()

**$x()** Returns an Array that Matches the XPath -> $x(“html/body/div”)

**State Data Expressions 🡪** We can view the state of ítems in a Ad Hoc Manner. The console evaluates any JS expression we type and gives Name Suggestions.

**More Chrome Dev Tools**

**Performance 🡪** Expectations of Code, Runtime Surprises, Detection of Problem Root.

We have Tools such as Timeline Recording and Profiler to get more detailed information. ( Performance Tab )

Recording: Overview -> Screenshots of our activity

Summay: IDLE should take most of the time.

Event Log: We see all the events that took place and it will notice when ther eis a performance issue.

**Networks 🡪** Network Dependent Code (All browsers), Unexpected Execution Issues, Proper Tooling.

Disable Cache: This gives us consistency that the cache won’t be used to see “first time characteristics” when loading the page.

Slow 3g: This will be the worst case scenario, so if our code runs fine there. Its an OK code.

CTRL + R: Will execute the Network and show the information on Time, Performance, Screenshots, etc.

**Memory Managment 🡪** Js Memory Manager, Additional Libraries and Assumptions, Proactive Debugging.

Chrome Task Manager: Browser ->Options -> More Tools -> Task Manager.

Memory: Represents the Native memory, including DOM Nodes. If it is increasing, DOM nodes are being created.

JavaScript Memory: Javascript HEAP, nos interesa el Number in paréntesis. Si el numero crece están siendo creados New Objects or the current Objects are increasing in size.

Performance Memory: Click Memory and Collect Garbage. We can see the JS HEAP and have the separated use of Memory.

\*\* If we see the JS HEAP increasing pay attention\*\*

Allocation