Examples of Web Workers

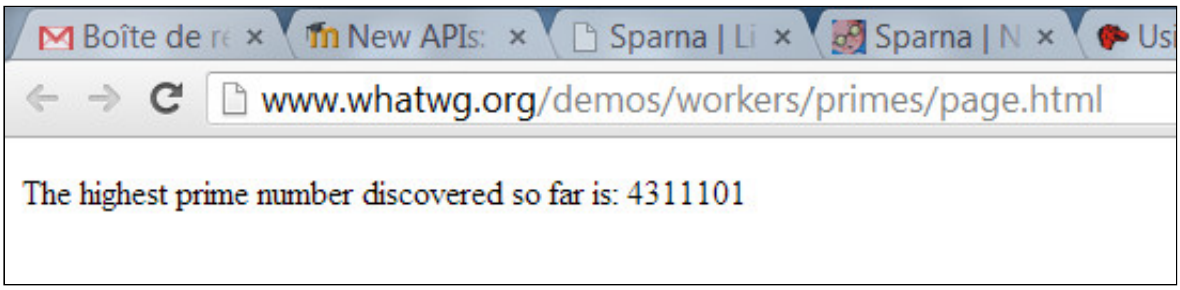
Dedicated Workers are the simplest kind of Workers. Once created, they are linked to their parent page (the HTML5 page that created them). There is an implicit "communication channel" opened between the Workers and the parent page, so that messages can be exchanged.

FIRST EXAMPLE: COMPUTE PRIME NUMBERS IN THE BACKGROUND WHILE KEEPING THE PAGE USER INTERFACE RESPONSIVE

Let's look at the first example, taken from the W3C specification: "*The simplest use of workers is for performing a computationally expensive task without interrupting the user interface. In this example, the main document spawns a worker to (naïvely) compute prime numbers, and progressively displays the most recently found prime number*."

This is the example we tried earlier, without Web Workers, that froze the page. This time it uses Web Workers. Notice that, unlike the previous example, it will display the prime numbers it has computed at regular intervals.

Try this example online (we cannot put it on JsBin as Workers need to be defined in a separate JavaScript file) :



The HTML5 page code from this example that uses a Web Worker:

```
<!DOCTYPE HTML>
<html>
 <head>
```

```
    <title>Worker example: One-core computation</title>
    </head>
    <body>
    <p>The highest prime number discovered so far
is: <output id="result"></output></p>
    <script>
        var worker = new Worker('worker.js');
10.     worker.onmessage = function (event) {

document.getElementById('result').textContent= event.data;
    };
    </script>
    </body>
</html>
```

In this source code, the Web Worker is created at *line 9*, and its code is in the `worker.js` file. *Lines 10-12* process a message sent asynchronously by the worker: `event.data` is the message content. Workers can only communicate with their parent page using messages. See the code of the worker below to see how the message has been sent.

The code of the worker (worker.js):

```
var n = 1;
search: while (true) {
    n += 1;
    for (var i = 2; i <= Math.sqrt(n); i +=1)
    if (n % i == 0)
    continue search;
    // found a prime!
    postMessage(n);
}
```

There are a few interesting things to note here:

1. There is an infinite loop in the code at *line 2* (while true...). This is not a problem as

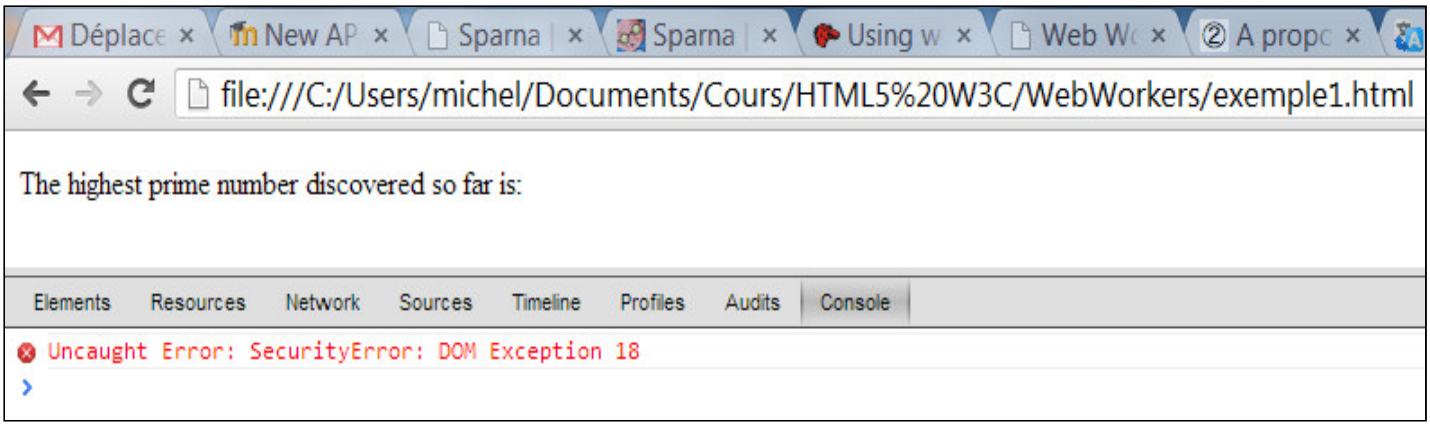
this code is run in the background.

2. When a prime number is found, it is "posted" to the creator of the Web Worker (aka the HTML page), using the`postMessage(...)` function.
3. Computing prime numbers by using such a bad algorithm is very CPU intensive. However, the Web page is still responsive; you can refresh it, the dialog "script not responding" does not appear, etc. There is a demo at the end of this chapter in which some graphic animation has been added to this example, and you can verify that the animation is not affected by the computations in the background.

TRY AN IMPROVED VERSION OF THE FIRST EXAMPLE YOURSELF

We can improve this example a little by testing whether the browsers support Web Workers, and displaying some additional messages.

CAREFUL: for security reasons you cannot try the examples using a file:// URL. **You need an HTTP web server that will serve your files**. Here is what happens if you do not follow this constraint:



This occurs with Opera, Chrome and Firefox. With Chrome, Safari or Chromium, you can run the browser using some command line options to override these security constraints. Read, for example,this blog post that explains this method in detail.

Ok, back to our improved version! This time, we test if the browser supports Web Workers, and we also use a modified version of the worker.js code for displaying a message, and wait 3 seconds before starting the computation of prime numbers.

You can download this example: WebWorkersExample1.zip

HTML code:

```
<!DOCTYPE HTML>
<html>
 <head>
 <title>Worker example: One-core computation</title>
 </head>
 <body>
 <p>The highest prime number discovered so far
is: <output id="result"></output></p>
 <script>
    if(window.Worker){
        // web workers supported by the browser
        var worker=new Worker("worker1.js");
        worker.onmessage=function(event){

document.getElementById('result').textContent= event.data;
        };
    }else{
        // the browser does not support web workers
        alert("Sorry, your browser does not support Web
Workers");
    }
 </script>
 </body>
</html>
```

Line 9 shows how to test if the browser can run JavaScript code that uses the HTML5 Web Workers API.

worker1.js code:

```
postMessage("Hey, in 3s, I'll start to compute prime
numbers...");
setTimeout(function() {
    // The setTimeout is just useful for displaying the
message in line 1 for 3 seconds and
```

```
        // making it visible
        var n = 1;
        search: while (true) {
            n += 1;
            for (var i = 2; i <= Math.sqrt(n); i+= 1)
10.         if (n % i == 0)
            continue search;
            // found a prime!
            postMessage(n);
         }
    }, 3000);
```

In this example, we just added a message that is sent to the "parent page" (*line 1*) then the standard JavaScript method `setTimeout()` is used for delaying the beginning of the prime number computation by 3s.

SECOND EXAMPLE: HOW TO STOP/KILL A WORKER AFTER A GIVEN AMOUNT OF TIME

So far we have created and used a worker. Now we will see how to kill it!

A worker is a thread, and a thread uses resources. If you no longer need its services, *it is best practice to release the used resources,*especially since some browsers may run very badly when excessive memory consumption occurs. *Even if we unassign the variable that was used to create the worker, it continues to live,* it does not stop! Worse: the worker becomes inaccessible but still exists (and therefore the memory is still used), and we cannot do anything but close the tab/page/browser.

The Web Worker API provides a `terminate()` method that we can use on any worker, ending its life. After a worker has been killed, it is not possible to undo its execution. The only way is to create a new worker.

HTML code:

```
<!DOCTYPE HTML>
<html>
 <head>
```

```
 <title>Worker example: One-core computation</title>
 </head>
 <body>
 <p>The highest prime number discovered so far
is: <output id="result"></output></p>
 <script>
    if(window.Worker){
       // web workers supported by the browser
       var worker=new Worker("worker2.js");
       worker.onmessage=function(event){

document.getElementById('result').textContent= event.data;
       };
    }else{
       // the browser does not support web workers
       alert("Sorry, your browser does not support Web
Workers");
    }
    setTimeout(function(){
       // After 10 seconds, we kill the worker
       worker.terminate();

document.body.appendChild(document.createTextNode("Worker
killed, 10 seconds elapsed !")
    );}, 10000);
 </script>
 </body>
</html>
```

Notice at *line 22* the call to `worker.terminate()`, that kills the worker after 10000ms.

`worker2.js` is the same as in the last example:

```
postMessage("Hey, in 3s, I'll start to compute prime
numbers...");
setTimeout(function() {
     // The setTimeout is just useful for displaying the
message in line 1 for 3 seconds and
```

```
        // making it visible
        var n = 1;
        search: while (true) {
            n += 1;
            for (var i = 2; i <= Math.sqrt(n); i+= 1)
            if (n % i == 0)
            continue search;
            // found a prime!
            postMessage(n);
        }
    }, 3000);
```

A WEB WORKER CAN INCLUDE EXTERNAL SCRIPTS

External scripts can be loaded by workers using the`importScripts()` function.

`worker.js`:

```
importScripts('script1.js');
importScripts('script2.js');
// Other possible syntax
importScripts('script1.js', 'script2.js');
```

The included scripts must follow the same-origin policy.

The scripts are loaded synchronously and the function`importScripts()` doesn't return until all the scripts have been loaded and executed. If an error occurs during a script importing process, a `NETWORK_ERROR` is thrown by the `importScripts`function and the code that follows won't be executed.

LIMITATIONS OF WEB WORKERS

Debugging threads may become a nightmare when working on the same object (see the "thread security" section at the beginning of this page). To avoid such a pain, the Web Workers API does several things:

1. When a message is sent, it is *always a copy* that is received: no more thread security problems.
2. Only predefined thread-safe objects are available in workers, this is a subset of those usually available in standard JS scripts.

Objects available in Web Workers:

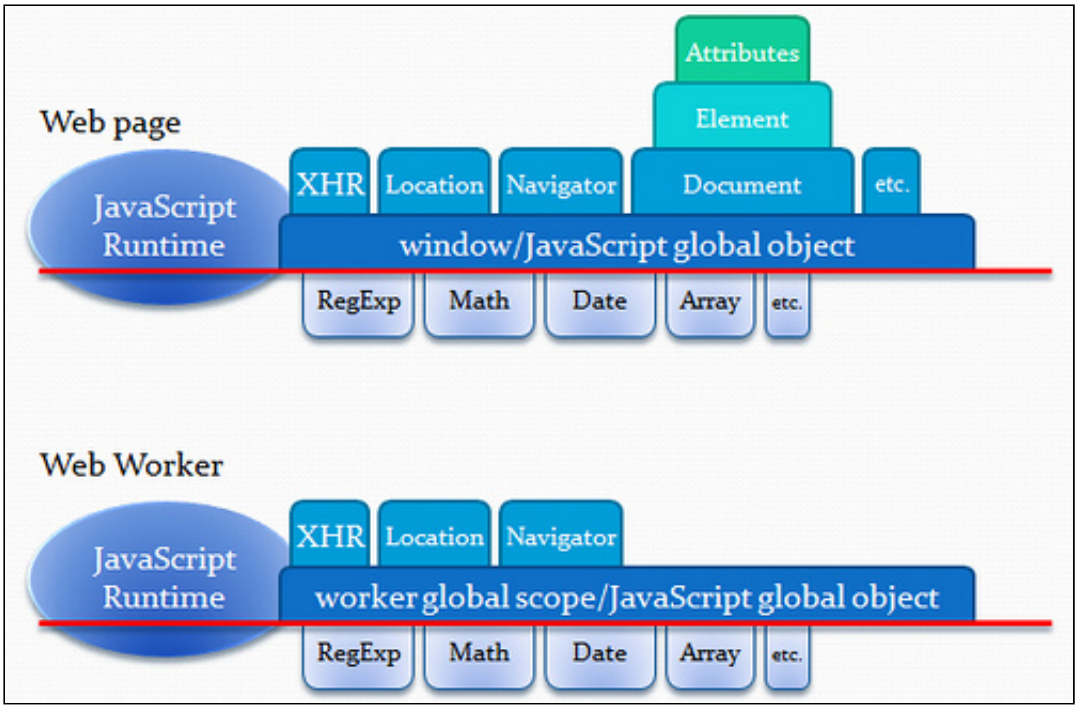
- The `navigator` object
- The `location` object (read-only)
- `XMLHttpRequest`
- `setTimeout()/clearTimeout()` and`setInterval()/clearInterval()`
- The Application Cache
- Importing external scripts using the `importScripts()` method
- Spawning other Web Workers

Workers do NOT have access to:

- The DOM (it's not thread-safe)
- The `window` object
- The `document` object
- The `parent` object

WOW! This is a lot! Son, please be careful!

We borrowed these two figures from a MSDN blog post (in French), as they illustrate this very well:



Note that:

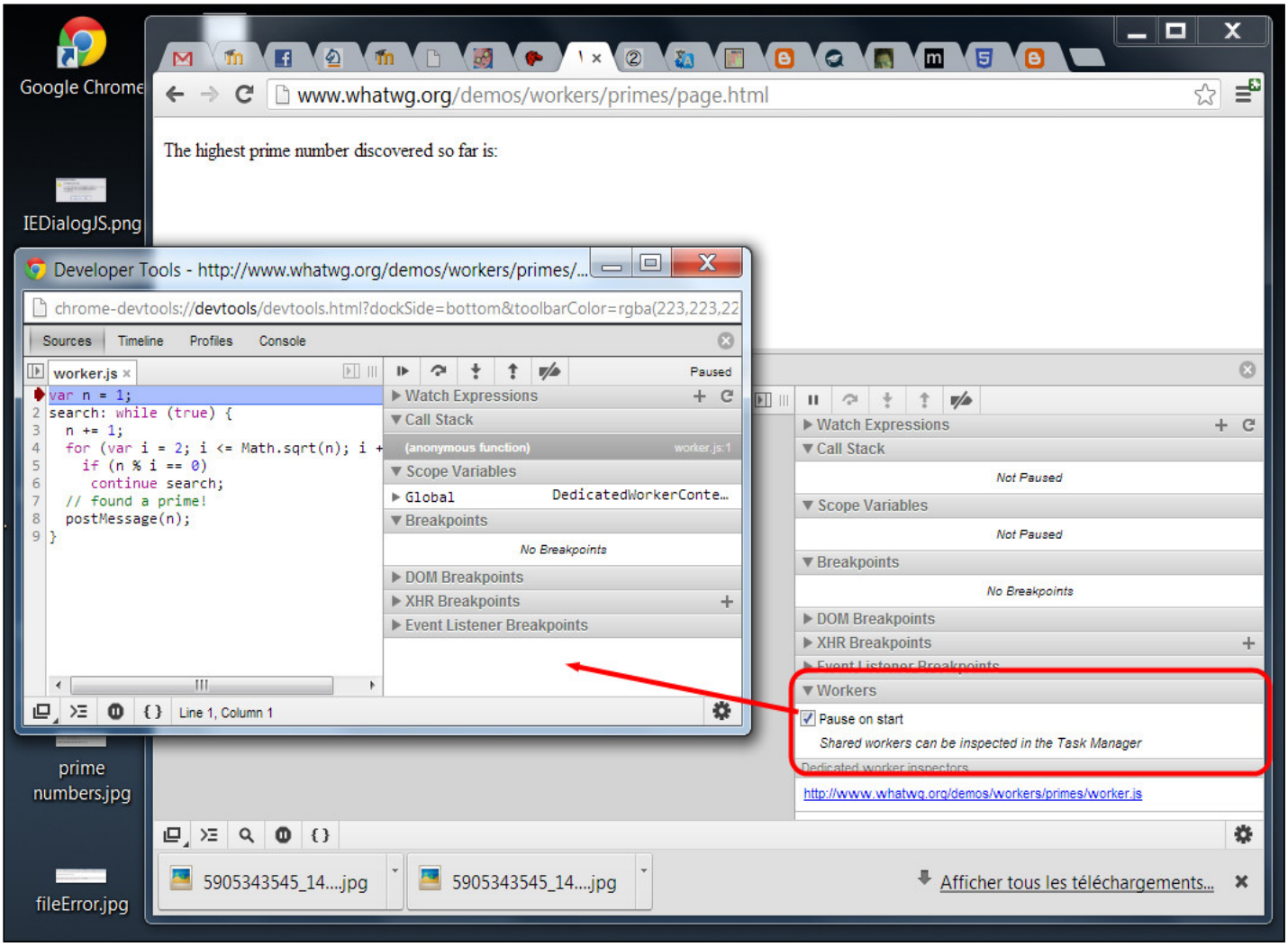
1. Chrome has already implemented a new way for transferring objects from/to Web Workers by reference, in addition to the standard "by copy" method. This is in the HTML 5.1 draft specification from the W3C - look for "transferable" objects!
2. The canvas is not usable from Web Workers, however, HTML 5.1 proposes a canvas proxy.

DEBUGGING WEB WORKERS

Like other multi-threaded applications, debugging Web Workers can be a tricky task, and having good instruments makes this process much easier.

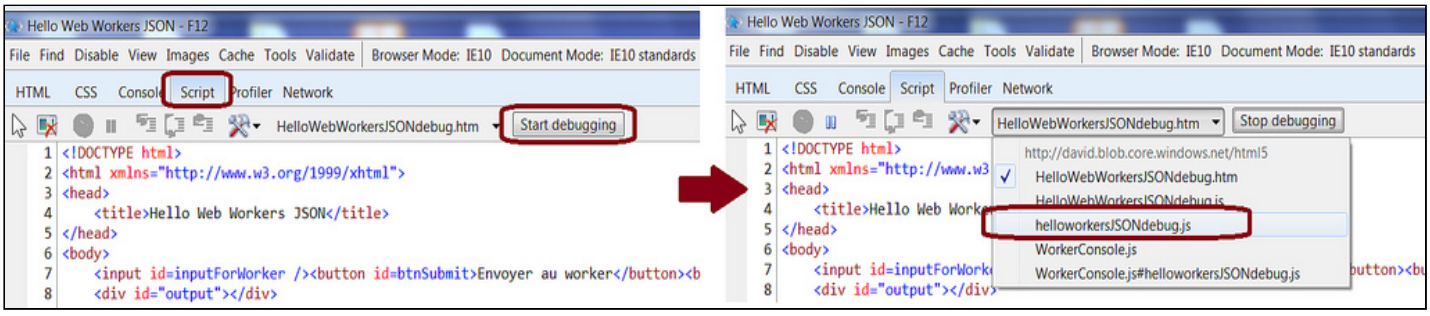
- Chrome provides tools for debugging Web Workers: read this post for details.

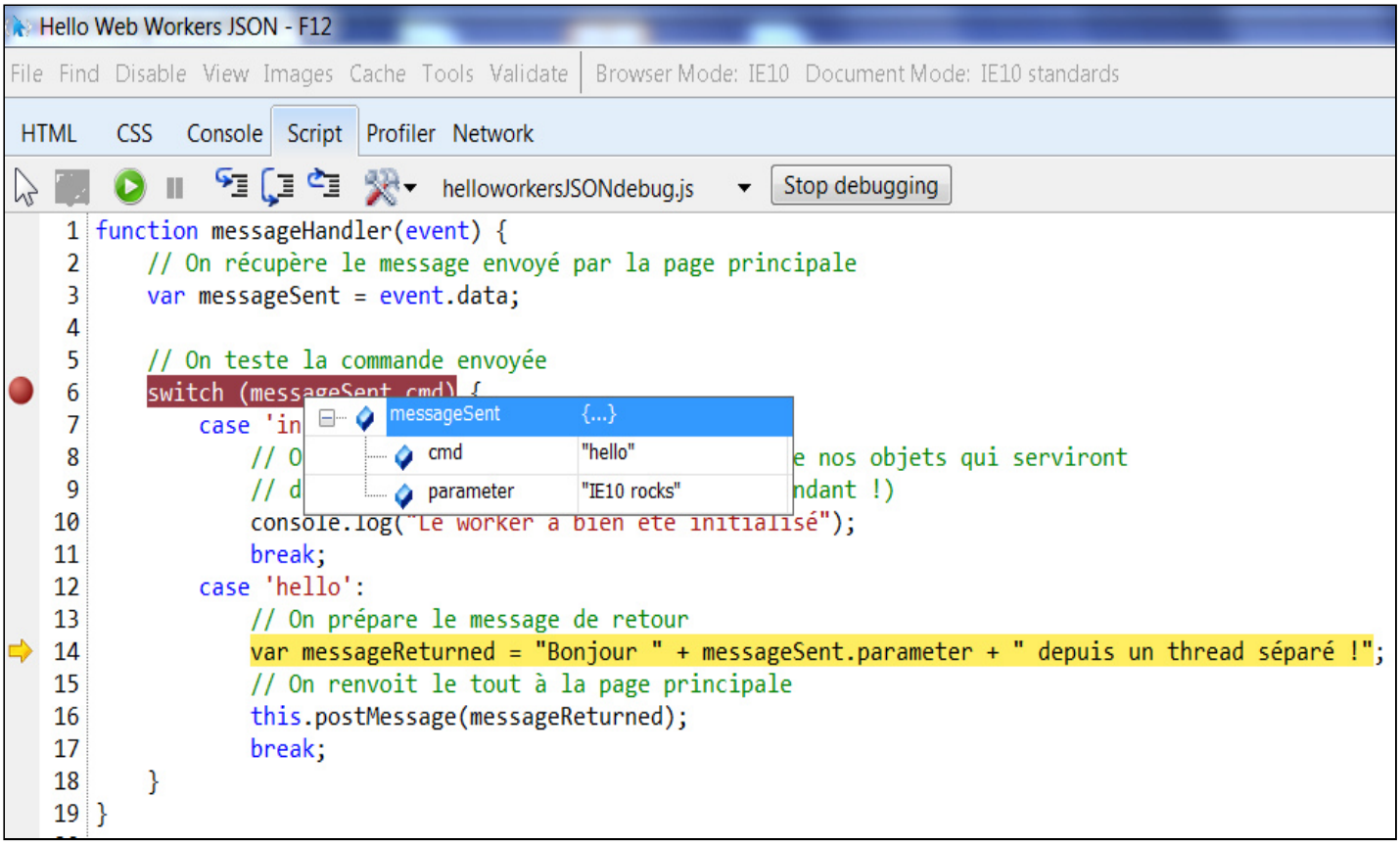
When you open a page with Web Workers, open the Chrome Dev Tools (F12), look on the right at the Workers tab, check the radio box and reload the page. This will pop a small window for tracing the execution of each worker. In these windows you can set breakpoints, inspect variables, log messages, etc. Here is a screenshot of a debugging session with the prime numbers example:



- IE 11 has some interesting debugging tools, too:

See this MSDB blog post for details. Load a page with Web Workers, press F12 to show the debugging tools. Here is a screenshot of a debugging session with Web Workers in IE11:





- FireFox has similar tools, see https://developer.mozilla.org/en-US/docs/Tools. For Operalook at the the Dragonfly documentation page, the Opera built in debugger.