Course description

Arnauld Van Muysewinkel
<avm@pendragon.be>
version 0.1, 07-Dec-2016
Initial version

Table of Contents

1. Target audience
2. Prerequisites
3. Training objectives
4. Course content
5. Skills acquired during the training
6. Out-of-scope
7. Approach

1. Target audience

Architects, Developers

2. Prerequisites

- Minimum 2 years of experience in developing Web applications (good knowledge of client-side technologies: HTML/CSS/JS)

3. Training objectives

The constant growth of mobile and responsive applications, the realization of applications increasingly rich and massively JavaScript, users demanding high standard response times, are all elements requiring to optimize the performance of web applications and websites.

The objective of this training course is to provide the best practices and tools necessary for this optimization.

4. Course content

- Why should we optimize websites?
- Rule for optimizing web sites and frequent issues
- Diagnostic tools - workshop
- Analysis and profiling tools - workshop

5. Skills acquired during the training

- Diagnose performance issues in the browser
- Identify and apply best practices and rules to make web applications efficient

6. Out-of-scope

- Performance analysis and benchmarking tools (see *T163 - Java Performance Tuning* (JPT) training).
- Best practices for Java Programming (see *T164 - Good Programming Practices for Programming* (GPP4P) training).

7. Approach

Theory + hands on

Version 0.1
Last updated 2016-12-07 09:12:18 CET

Training Plan

Arnauld Van Muysewinkel
<avm@pendragon.be>
version 0.1, 07-Dec-2016
Initial version

Table of Contents

1. Introduction
2. Preamble
3. Transport
4. Processing
5. Other courses

1. Introduction

- Course description
- Introduction
- Objectives

2. Preamble

- Performance improvement process
- Web Browser architecture

3. Transport

- Tuning transport issues

4. Processing

- CPU (compilation)
- Memory (garbage collector)

5. Other courses

" Java Performance Tuning "

- Performance benchmarking
- Performance Diagnostic Model
- Performance testing process

" Good Programming Practices for Performance "

- Best practices to avoid performance issues
  - Infrastructure: CPU architecture, JIT compiler…
  - Middleware: JSF, JPA, SQL…
- Micro-profiling

Version 0.1
Last updated 2016-12-07 09:12:18 CET

Introduction

Arnauld Van Muysewinkel
<avm@pendragon.be>
version 0.1, 07-Dec-2016
Draft version

Table of Contents

1. Content
2. Why should we optimize websites?
3. Scope
4. Exercises
5. Disclaimer

1. Content

- Introduction

(back to course plan)

2. Why should we optimize websites?

- Improve user experience
- Improve page ranking (SEO)
- Mobile compliance
- UI logic is shifting to the front-end ⇒ increasing complexity

Some figures:

- 94.4% of web sites use JavaScript (as of 2016-Dec)
- users abandon a web site after 2s of waiting
- 90% of the waiting time is spent **after** the HTML has been retrieved
- 80% of the waiting time is spent **in** the front-end

3. Scope

"Transport" section:

- all kinds of web apps

"CPU" and "Memory" sections:

- Web apps with browser side logic
  - traditional (eg JSF) with "smart" components, AJAX...
  - SPA (Single Page Apps)
- i.e. applications relying heavily on JavaScript

Out-of-scope:

- animations, games...

4. Exercises

Exercises will be based on:

- the Chrome browser,
- Blink(WebKit) rendering engine,
- V8 JavaScript engine.

5. Disclaimer

- We’re still learning
- Landscape evoluates extremely rapidly
- Optimizing a web app is more an art than a science

Objectives

Arnauld Van Muysewinkel
<avm@pendragon.be>
version 0.1, 07-Dec-2016
Draft version

Table of Contents

1. Content
2. Objectives of the course

1. Content

- objectives of the course

(back to course plan)

2. Objectives of the course

TODO

Performance improvement process

Arnauld Van Muysewinkel
<avm@pendragon.be>
version 0.2, 11-May-2017
Draft version

Table of Contents

1. Content
2. Useful references
3. General recommendations
4. Improvement process
5. Performance test
6. Monkey testing
7. Other topics

1. Content

- General recommendations
- Improvement process

(back to plan)

2. Useful references

- Devoxx France : "Optimiser les performances d’une webapp" (Guillaume EHRET)
  https://www.youtube.com/watch?v=9PRPPJFaF_o

3. General recommendations

- no premature optimization
- must have a performance objective
- should have a workbench (performance test)
- "80-20": concentrate of the 20% of the app that is responsible for 80% of the load
- one issue often hides other issues
- first actions might not show any improvement.
  Yet, don’t despair!

4. Improvement process

- run test
- measure
- if performance objective not reached:
  - analyse results, look for:
    - transport issues,
    - then GC issues,
    - then CPU/code optimization
  - fix (one at a time!)
  - iterate (→ run test)

5. Performance test

Test data

- test with various parameters, especially corner cases

Test harness

- properly isolated
- test in real-life conditions:
    - network (bandwidth and latency)
    - server load (does impact response time)
    - browser brand and version
    - client device type (desktop, mobile phone…) and sizing (CPU, RAM…)

Test scenario

- representative of the issue

6. Monkey testing

- Lot’s of random actions (automated)

→ objective is to make sure there is no way the app could crash

(see http://www.redotheweb.com/2014/01/07/completing-the-js-test-stack-introducing-greminsjs.html) → https://github.com/marmelab/gremlins.js

7. Other topics

http://superherojs.com/#performance

Browser architecture

Arnauld Van Muysewinkel
<avm@pendragon.be>

version 0.2, 11-May-2017
Draft version

Table of Contents

1. Content
2. References
3. Web Browser
4. Chrome
5. Firefox
6. Internet Explorer (pre-Edge)
7. Rendering Engine
8. Blink (WebKit)
9. Other rendering engines
10. JavaScript Engine
11. Components of a JavaScript Engine
12. JavaScript Engine: flow of processing
13. V8
14. V8: Interpreter
15. V8: Compilers
16. V8: Ignition pipeline
17. V8: Garbage Collector
18. V8: Orinoco GC
19. Other JavaScript engines

1. Content

- Web Browser
    - Chrome
    - Firefox
    - Internet Explorer
- Rendering Engine
    - Blink (WebKit)
    - Others
- JavaScript Engine
    - V8
    - Others

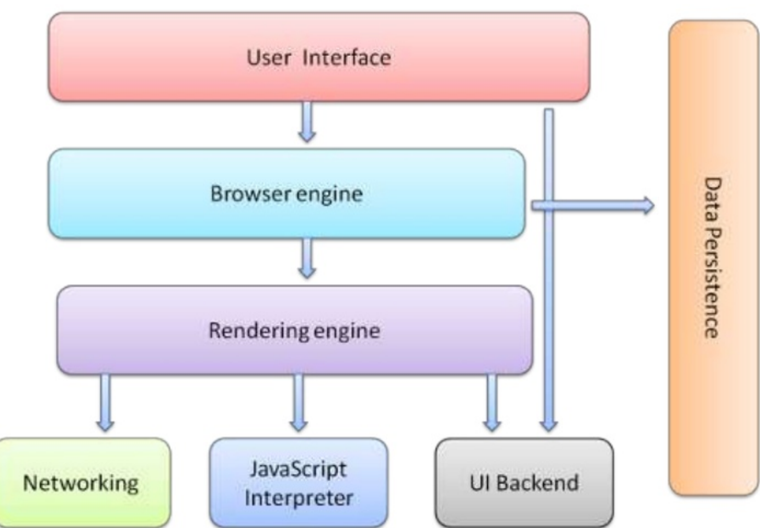
(back to plan)

2. References

- A Reference Architecture for Web Browsers. Alan Grosskurth & Michael W. Godfrey. University of Waterloo. http://grosskurth.ca/papers/browser-refarch.pdf
- How Browsers Work - Part 1 - Architecture. Vineet Gupta. http://archive.li/YOGPn
- Web-Browser Architecture. Nguyen Quang. Saltlux. 2015-05. http://www.slideshare.net/quangntta/web-browser-architecture-49196378
- Mobile Browser Internals. (Understanding Blink Rendering Engine). Hyungwook Lee. Naver Labs. 2014-01. http://www.slideshare.net/HyungwookLee/mobilebrowserinternal-20140122
- Official blog of the V8 JavaScript engine. http://v8project.blogspot.be/
    - V8 Release 5.6. 2016-12-02. http://v8project.blogspot.be/2016/12/v8-release-56.html
- Implementing a JavaScript Engine. Kris Mok. Azul Systems. 2013-11-10. http://www.slideshare.net/RednaxelaFX/implement-js-krystalmok20131110
- How Browsers Work: Behind the scenes of modern web browsers. Tali Garsiel & Paul Irish. HTML5 Rocks Tutorials. 2011-08-05. https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/

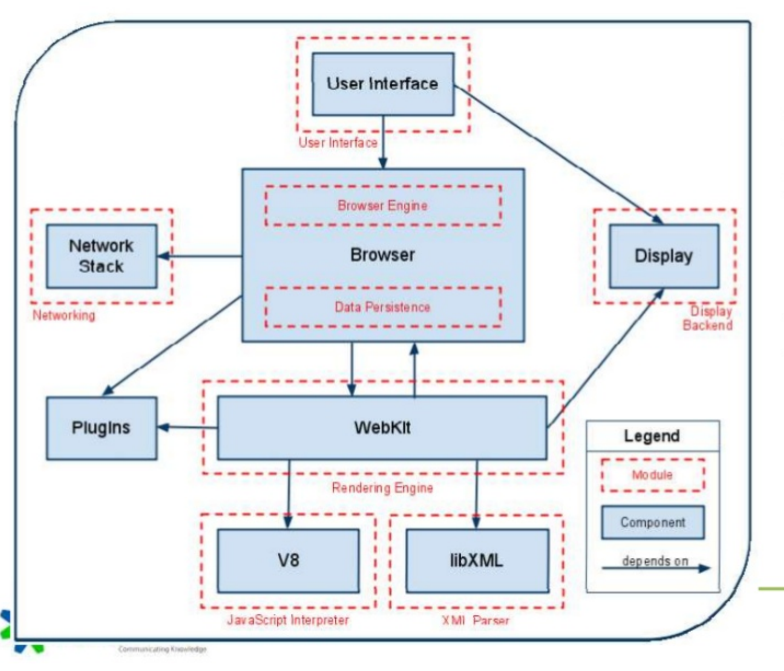
3. Web Browser

The browser's main components



4. Chrome

Architecture of Chrome



Rendering Engine:

Used the WebKit until v27, from v28 user WebKit fork Blink

XML Parser:

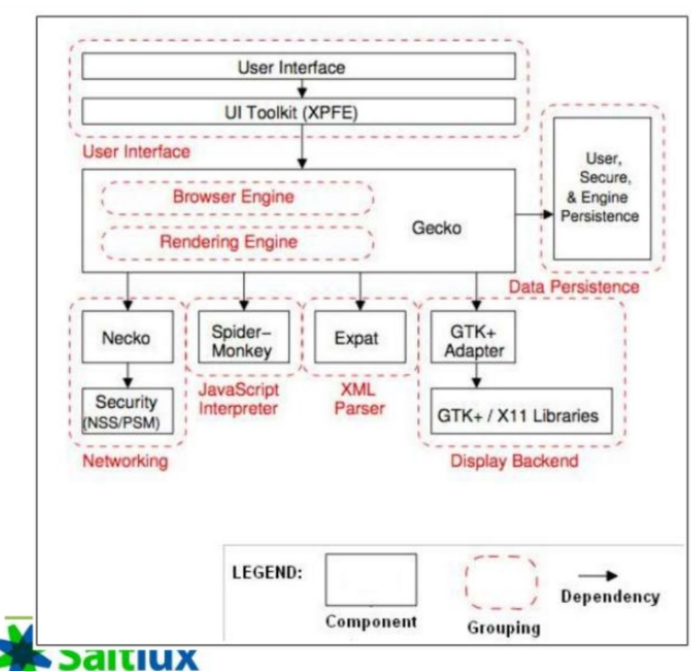
libXML to parse XML

libXSLT to handle XSLT

JavaScript Interpreter: V8 JavaScript Engine, writen in C++

5. Firefox

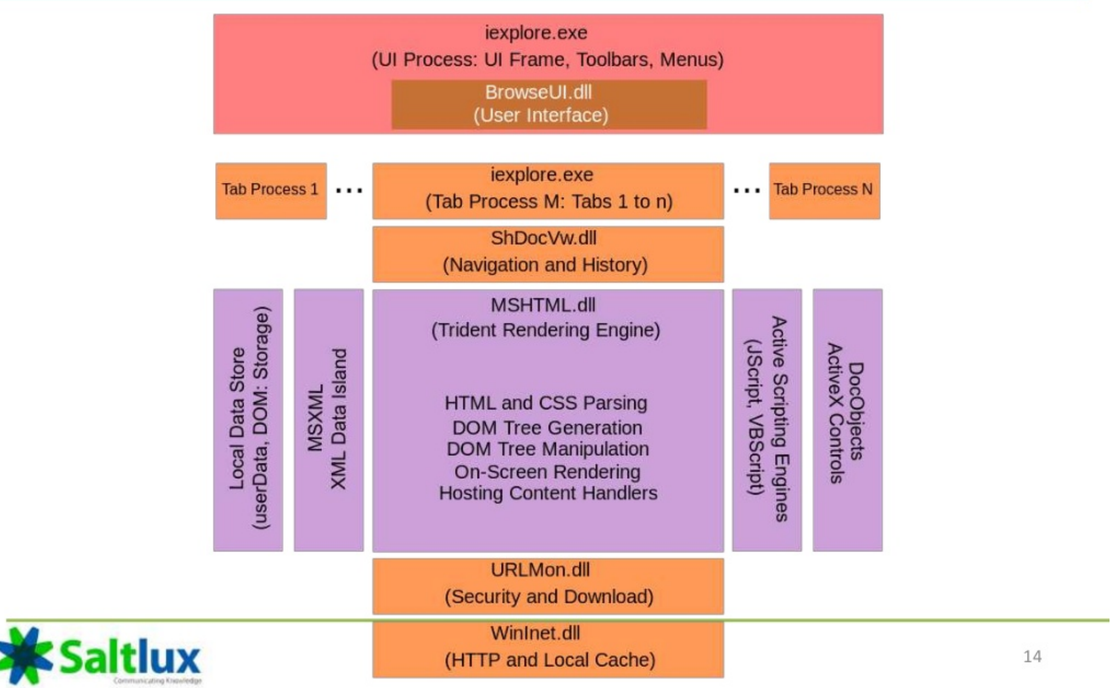
Architecture of Firefox



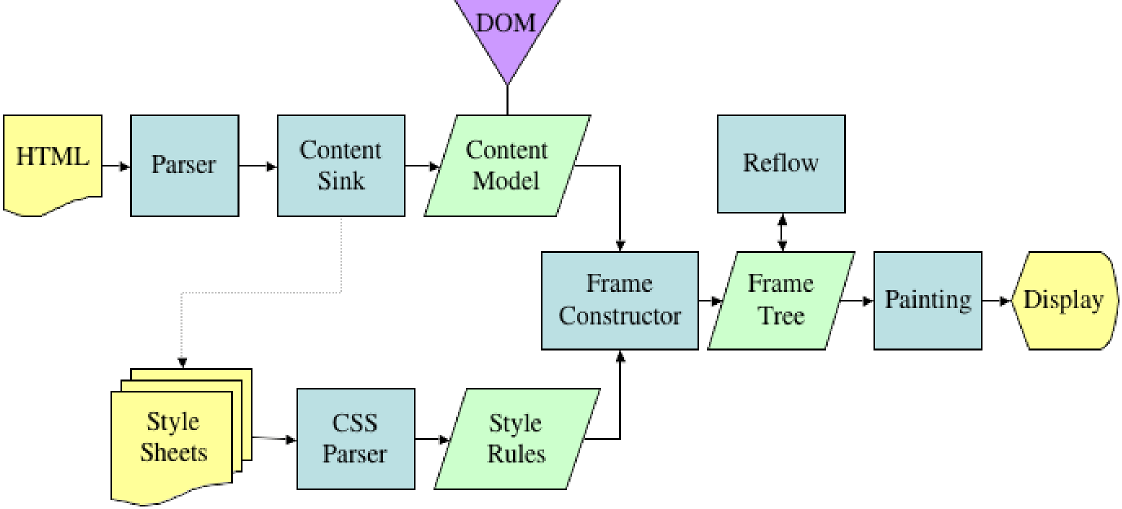
Rendering Engine: Gecko
XML Parser: Expat
JavaScript Interpreter: Spider-Monkey, implement in C

6. Internet Explorer (pre-Edge)

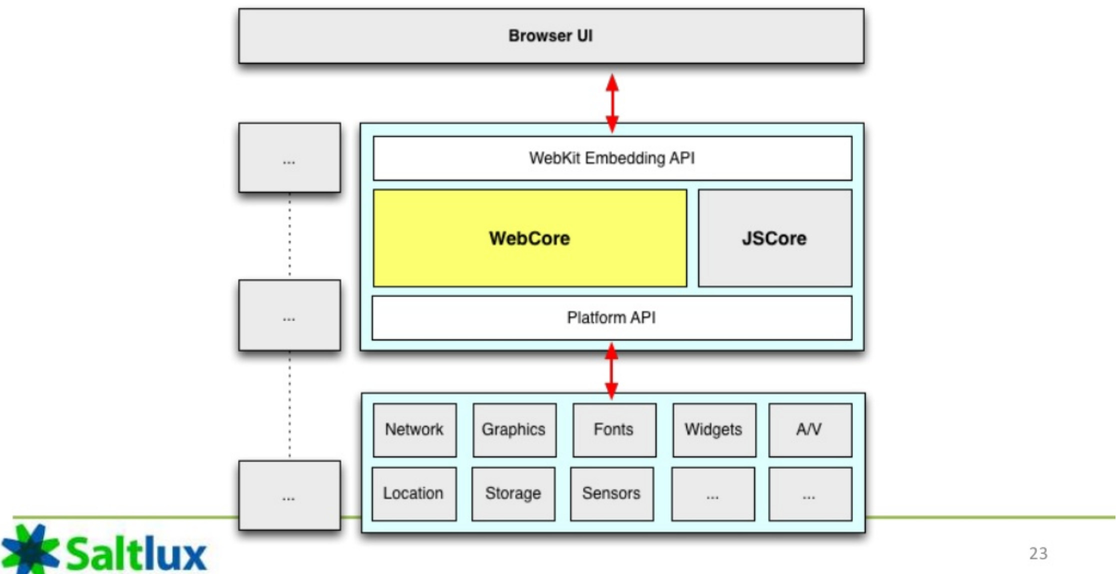
Architecture of IE



7. Rendering Engine



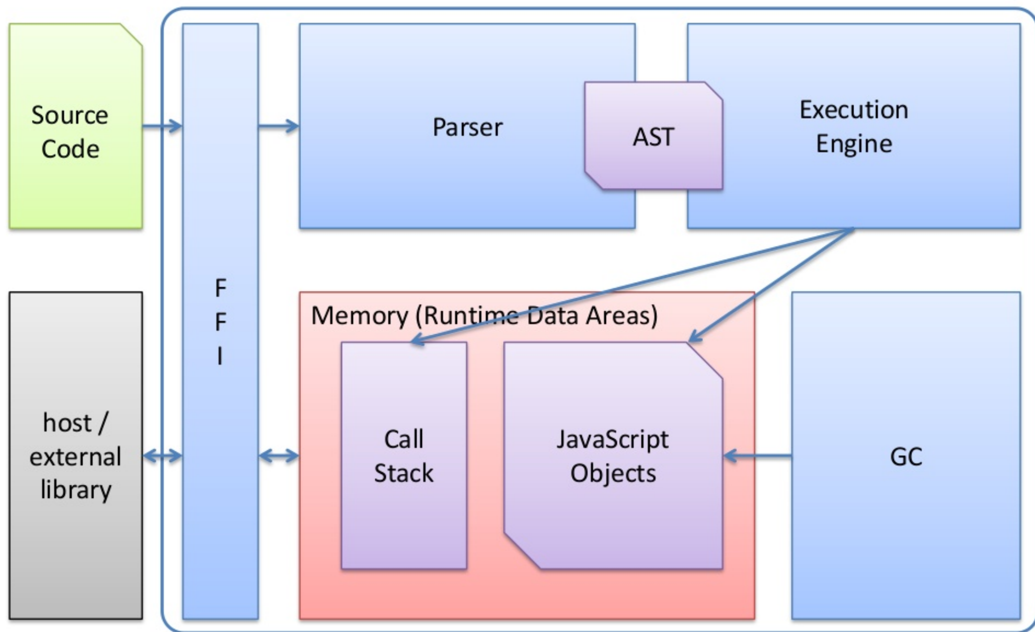
8. Blink (WebKit)



9. Other rendering engines

- Gecko (Firefox)
- Trident (Internet Explorer)
- EdgeHTML (Edge)
- WebKit (Safari)

10. JavaScript Engine



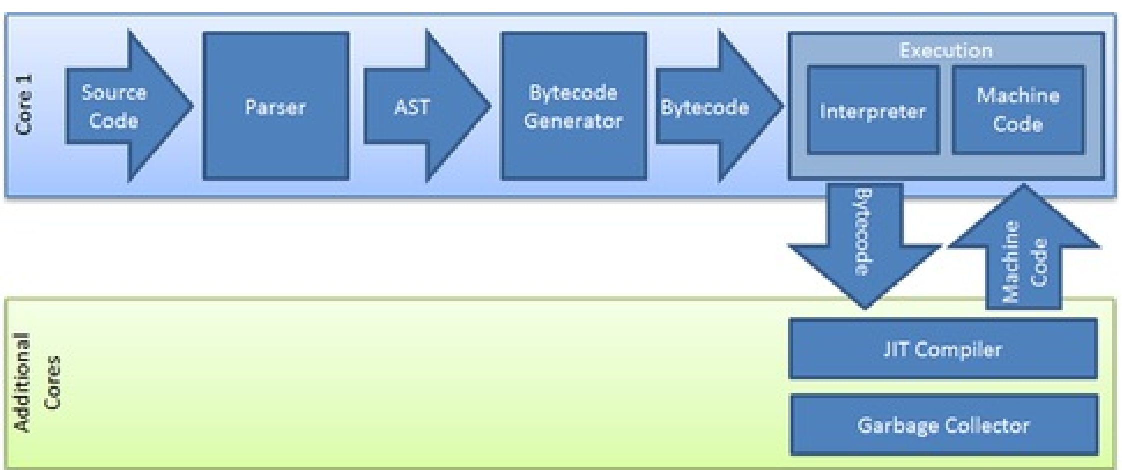
11. Components of a JavaScript Engine

AST (Abstract Syntax Tree)
: tree representation of the code structure

FFI (Foreign Function Interface)
: translates calls between 2 languages

GC (Garbage Collector)
: manages heap memory

12. JavaScript Engine: flow of processing



13. V8

https://github.com/v8/v8/wiki/Design%20Elements
http://thibaultlaurens.github.io/javascript/2013/04/29/how-the-v8-engine-works/
http://stackoverflow.com/a/12987881/318354
http://websrv0a.sdu.dk/ups/SCM/slides/lecture_03_mads_ager.pdf

14. V8: Interpreter

https://github.com/v8/v8/wiki/Interpreter > Design document

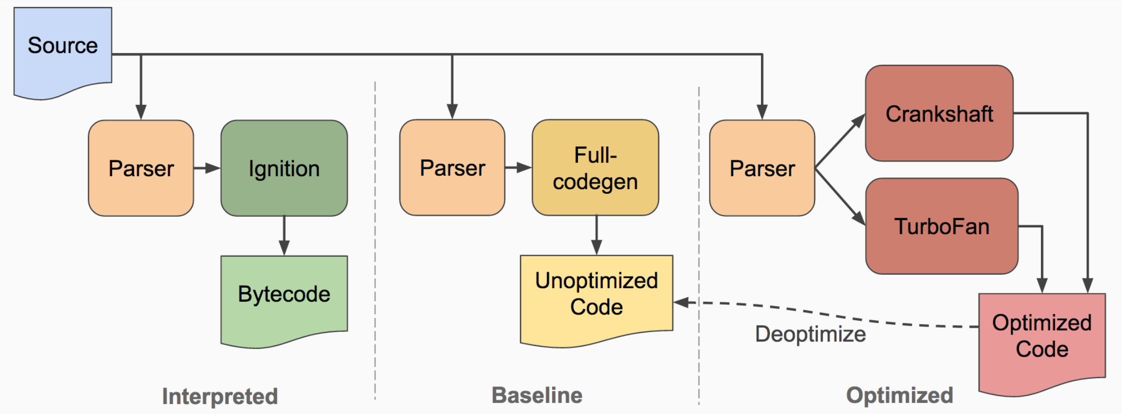
15. V8: Compilers

- baseline compiler "full-codegen" → quick-and-dirty
- 2010: + CrankShaft → focus on hot-spots
- 2015: + TurboFan → improved performance, asm.js

https://github.com/v8/v8/wiki/TurboFan http://wingolog.org/archives/2011/07/05/v8-a-tale-of-two-compilers

16. V8: Ignition pipeline

As of Dec-2016:



→ avoids memory overhead of the compilation

- JavaScript is first compiled to "concise" bytecode
- smaller than native code
- high performance interpreter delivers performance close to current compilers

17. V8: Garbage Collector

The problem:

- in general, determining if a piece of memory is used or not is an **undecidable problem**
- memory management may be **manual** (tedious and error prone) or **automatic**

Garbage Collection:

- a type of automatic memory management
- based on "Mark-and-Sweep" algorithms
  - start from a list of "roots"
  - browse the reference tree and mark each object as "active"
  - all other objects are inactive and may be reclaimed

This is similar to Java.

18. V8: Orinoco GC

Orinoco (since V8 5.0, Chrome 50)

Generational GC: Young/Old generations

- promotion after 2 generation (= "Evacuation")
- spaces partitioned in fixed size blocks ("pages")
- evacuation in parallel, on a page by page basis
- old compaction in parallel, on a page by page basis

Improved pointers tracking

- speeds up objects relocation

Black allocation

- speeds up marking of objects in old generation
  (promoted objects are considered active by default)

https://blog.risingstack.com/javascript-garbage-collection-orinoco/
http://v8project.blogspot.be/2016/04/jank-busters-part-two-orinoco.html

19. Other JavaScript engines

- Monkey (Gecko, Netscape/Mozilla/Firefox)
  - Spidermonkey (legacy): first ever JavaScript engine
  - TraceMonkey: first JIT compiler
  - JägerMonkey: improvement of TraceMonkey, superseded by IonMonkey
  - IonMonkey: current engine with Gecko
- Chakra (Trident, Internet Explorer)
- SquirrelFish (WebKit, Safari) (was: JavaScriptCore)

Tuning transport issues

Arnauld Van Muysewinkel
<avm@pendragon.be>
version 0.2, 14-May-2017
Draft version

Table of Contents

1. Content
2. Most frequent improvement axis
3. Reduce HTTP requests count
4. Use client-side processing
5. Use caches
6. Reduce sizes
7. (Use a Global Network)
8. Optimize depending on the context
9. Reduce first rendering latency
10. Optimize on the service side
11. Investigation tools
12. Timeline
13. Timeline
14. Boomerang
15. Online tools
16. Lighthouse
17. Progressive Web Apps (PWA)
18. HTTP/2
19. Other topics
20. Exercise: Elements order

1. Content

- Most frequest improvement axis
- Investigation tools
- Progressive Web Apps
- HTTP/2
- Exercises

(*back to plan*)

2. Most frequent improvement axis

- Reduce HTTP requests count
- Use client-side processing
- (Parallelize requests)
- Use caches
- Reduce sizes
- (Use a Global Network, or CDN (Content Delivery network))
- Optimize depending on the context
- Reduce first rendering latency
- Optimize on the server side

3. Reduce HTTP requests count

- combine small images into one
- combine several scripts as one
- combine several stylesheet as one
- avoid duplicate (or useless!) scripts

4. Use client-side processing

- validations
- ...

5. Use caches

- `Expires` header
  → date/time after which the response is considered stale
- `ETag` header
  → identifier for a specific version of a resource
- make scripts and stylesheets external

6. Reduce sizes

- minification of scripts and stylesheets
- compression of HTTP payload

Example: Angular.js

- minification: 729 KB → 101 KB
- gzip compression: 101 KB → 37 KB

7. (Use a Global Network)

- CDN (Content Delivery Network or Content Distribution Network)
- globally distributed network of proxy servers deployed in multiple data centers

Goal: Serve content to end-users with:

- high availability
- high performance

8. Optimize depending on the context

Context:

- device
- bandwidth

→ adapt size of resources (images)
→ adapt complexity of page

9. Reduce first rendering latency

- stylesheets at the top
- scripts at the bottom

10. Optimize on the service side

- Reverse Proxy (offloading of some tasks)
- Load Balancer
- Cache static files
- Compress data
- Tune your backend (OS, middleware, application) → see JPT course
- Monitoring

https://www.nginx.com/blog/10-tips-for-10x-application-performance/

11. Investigation tools

- Timeline
- Boomerang
- Online tools
- Lighthouse

12. Timeline

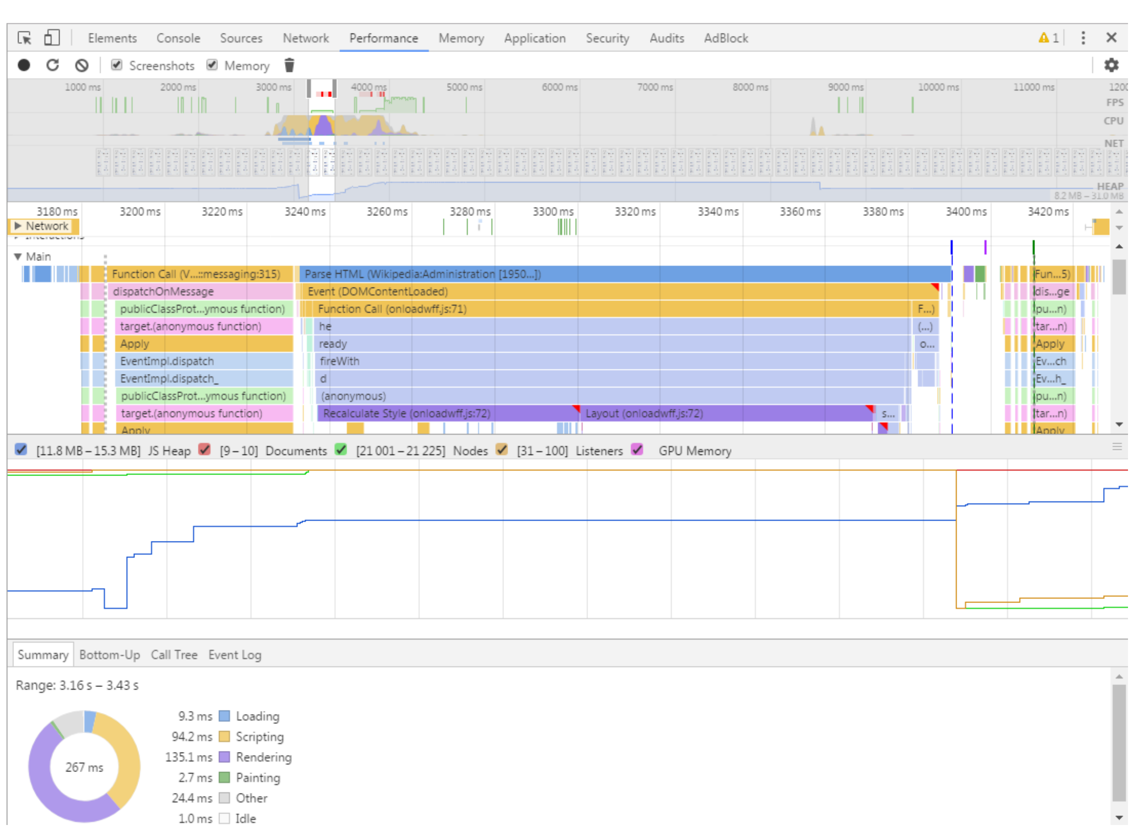
in Chrome:
>> Developer Tools >> **Performance** >> (Record button)

Recording of:

- **FPS** (Frame Per Second) (60 fps = 16.67 ms / frame)
- **CPU** resources: shows what type of events consumed the CPU resources
- **NET**: each bar represents an HTTP resource. The longer the bar, the slower the request-response
  - HTML files are blue.
  - Scripts are yellow.
  - Stylesheets are purple.
  - Media files are green.
  - Miscellaneous resources are grey.
- Paint events
- Screenshots
- Memory consumption (more on this later)
- JS Profile (more on this later)

https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/timeline-tool

13. Timeline



14. Boomerang

Performance monitoring from the backend

- piece of javascript added to web pages
- measures the performance from end users point of view
- sends this data back to the server

https://yahoo.github.io/boomerang/doc/

15. Online tools

Online application executing a number of predefined tests on a given web page.

Gives an overal evalution (/ 100 points) + detailed report with advices.

Only for applications visible publicly

https://developers.google.com/speed/pagespeed/insights

16. Lighthouse

- Chrome extension: https://github.com/GoogleChrome/lighthouse

→ verify criterias defined by the PWA specification (see further), and give a global evaluation (/ 100 points) + detailed report with advices

17. Progressive Web Apps (PWA)

→ Bring user experience of native apps to web apps

This is made possible by modern web technologies

According to Google, they must have the following characteristics:

progressive - responsive - work offline - app-like - fresh - safe - discoverable - re-engageable - installable - linkable

https://developers.google.com/web/progressive-web-apps/

18. HTTP/2

Reduce latencies through:

- Data compression of HTTP headers
- HTTP/2 Server Push
- Pipelining of requests
- Multiplexing multiple requests over a single TCP connection

→ a number of current practices should disappear

19. Other topics

- Guidelines REST (cf site SOA)

20. Exercise: Elements order

Disable the cache for this exercise: Developer Tools >> Network >> ☒ Disable cache

elementsOrder.html

```
<html>
<head>
    <title>Elements order</title>
<script src="https://code.jquery.com/jquery-3.2.1.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/dojo/1.12.2/dojo/dojo.js"></script>
<script src="https://code.angularjs.org/tools/system.js"></script>
<script src="https://code.angularjs.org/tools/typescript.js"></script>
<script src="https://code.angularjs.org/2.0.0-alpha.39/angular2.dev.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/phaser/2.6.2/phaser.js"></script>
</head>
<body>
Here is (finally!) the page...
</body>
<html>
```

CPU (compilation)

Arnauld Van Muysewinkel

<avm@pendragon.be>

version 0.2, 14-May-2017
Draft version

Table of Contents

1. Content
2. Typical causes of performance issues
3. Cause: Implementations
4. Cause: Hidden classes
5. Cause: Repetitions
6. Cause: Generic API
7. Causes...
8. Profiling
9. CPU Profiler
10. CPU Profiler
11. CPU Profiler
12. Timeline recording
13. Timeline recording
14. Other topics
15. Lower level languages
16. Exercise: Context promotion

1. Content

- Typical causes of performance issues
- Profiling
- Exercises

(back to plan)

2. Typical causes of performance issues

- Implementations
- Hidden classes
- Repetitions
- Generic API

3. Cause: Implementations

Choose API methods carefully. There are often several ways of doing something, but only one is most efficient.

For example, avoid:

```
... = str.split("'").join("\'");
```

prefer:

```
... = str.replace(/'/g, "\'");
```

4. Cause: Hidden classes

- classes generated internally by V8 to represent each object type
- all objects created with the same members use the same hidden class
- ⇒ avoid changing the type (i.e. adding members) of an object after construction

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

var p1 = new Point(11, 22);
var p2 = new Point(33, 44);
// At this point, p1 and p2 have a shared hidden class
p2.z = 55;
// warning! p1 and p2 now have different hidden classes!
```

5. Cause: Repetitions

Avoid calling expensive objects in a loop.

Example: create a regular expression once, *before* entering the loop

Avoid:

```
function on(events, ...) {
  events = events.split(/\s+/);
  ...
}
```

Prefer:

```
var eventSplitter = /\s+/;
function on(events, ...) {
  events = events.split(eventSplitter);
  ...
}
```

6. Cause: Generic API

More generic API's are usually slower.

For example, avoid:

```
arr.slice(n)[0] // with n < 0, returns the n-th to last element
```

prefer:

```
arr[arr.length + n] // with n < 0, does the same
```

(because `slice(n)` makes a partial copy of the array into another array)

7. Causes…

And many others...

8. Profiling

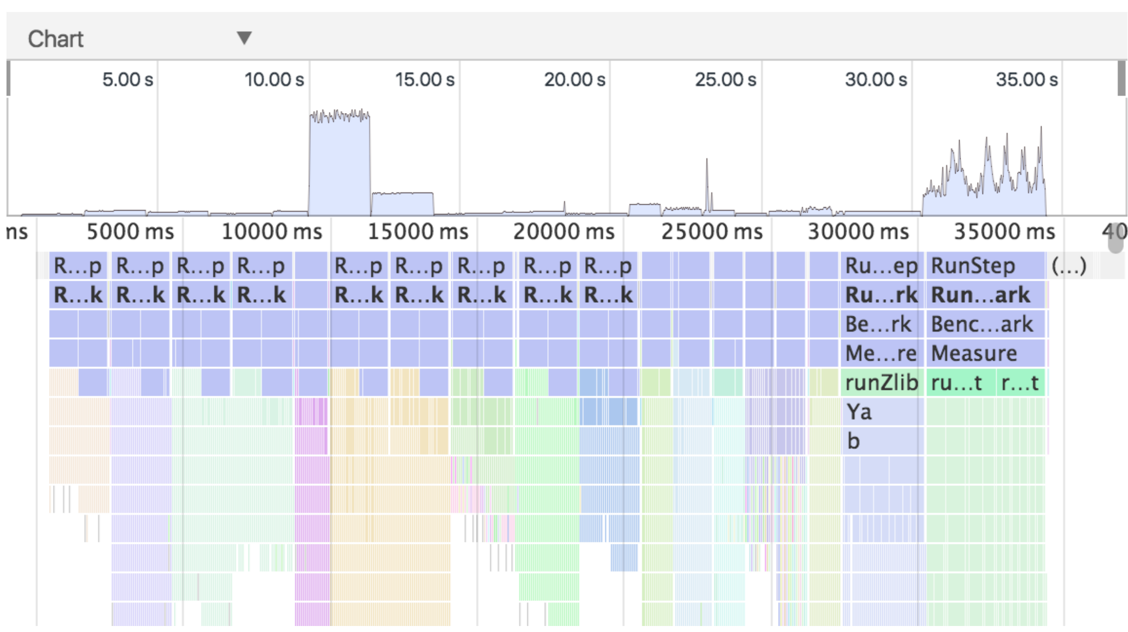
- CPU profiling
- Timeline recording

9. CPU Profiler

in Chrome:

>> Developer Tools >> ⋮ >> More tools >> JavaScript Profiler >> **Start**
...
>> Stop
Then, select "Chart" to get a *flame chart*.
Or, select "Heavy (Bottom Up)" to get histogram data.

10. CPU Profiler



11. CPU Profiler

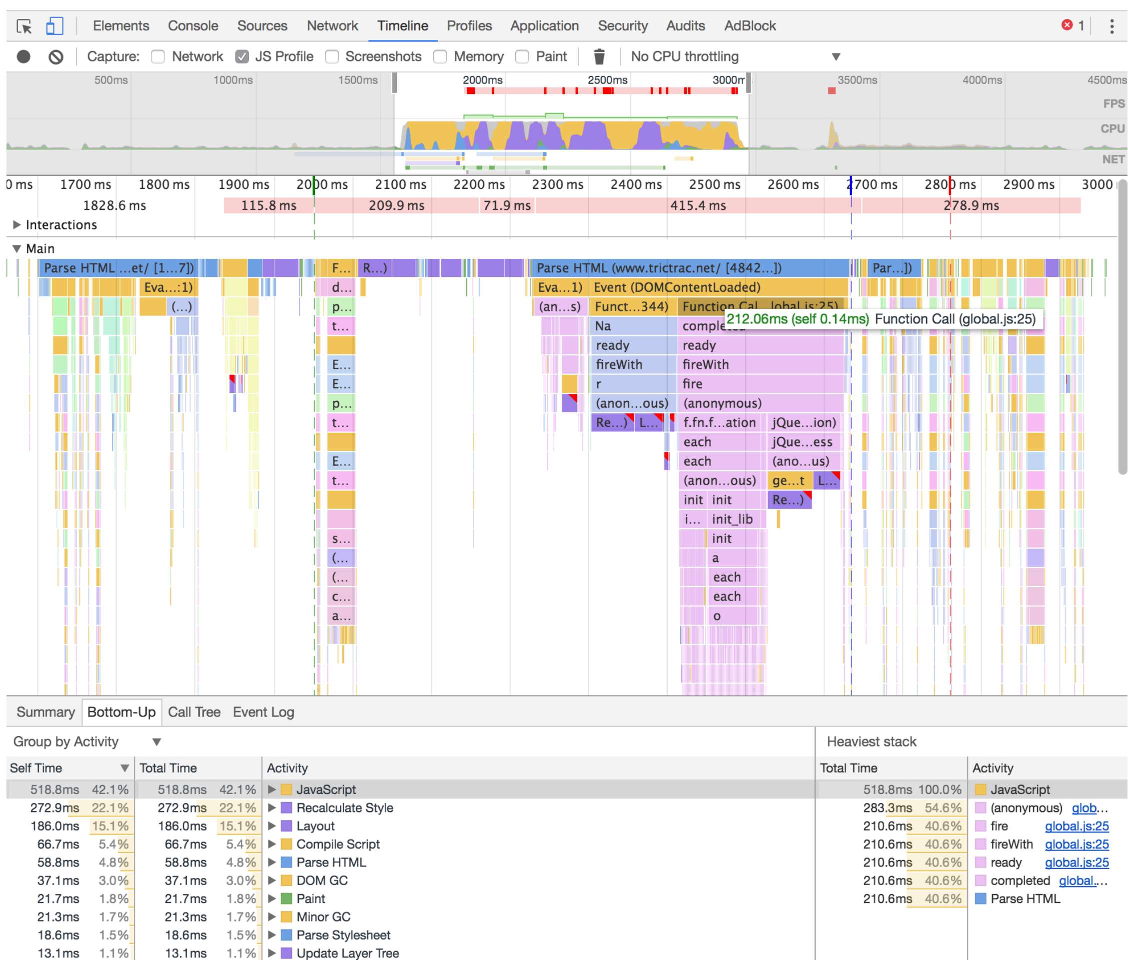
Heavy (Bottom Up) ▼

Self		Total		Function	
4447.3 ms		4447.3 ms		(idle)	
2162.6 ms	6.61 %	2165.4 ms	6.62 %	▶ montReduce	crypto.js:583
1951.8 ms	5.97 %	1951.8 ms	5.97 %	(garbage collector)	
1643.9 ms	5.02 %	1652.8 ms	5.05 %	▶ lin_solve	navier-stokes.js:152
1476.7 ms	4.51 %	1964.1 ms	6.00 %	▶ Scheduler.schedule	richards.js:188
1271.8 ms	3.89 %	1271.8 ms	3.89 %	(program)	
1170.8 ms	3.58 %	1172.0 ms	3.58 %	▶ bnpSquareTo	crypto.js:431
987.9 ms	3.02 %	1081.7 ms	3.31 %	▶ GeneratePayloadTree	splay.js:50
884.5 ms	2.70 %	2269.5 ms	6.94 %	▶ a8	(program):1
763.5 ms	2.33 %	837.0 ms	2.56 %	▶ one_way_unify1_nboyer	earley-boyer.js:3635
720.7 ms	2.20 %	720.7 ms	2.20 %	▶ a6	(program):1
682.6 ms	2.09 %	1577.2 ms	4.82 %	▶ rewrite_nboyer	earley-boyer.js:3604
624.5 ms	1.91 %	624.5 ms	1.91 %	▶ SplayTree.splay_	splay.js:322
619.2 ms	1.89 %	846.0 ms	2.59 %	▶ Exec	regexp.js:1
558.0 ms	1.71 %	795.0 ms	2.43 %	▶ (anonymous function)	code-load.js:1518
540.0 ms	1.65 %	540.4 ms	1.65 %	▶ Plan.execute	deltablue.js:776
517.8 ms	1.58 %	799.7 ms	2.44 %	▶ (anonymous function)	code-load.js:1541

12. Timeline recording

in Chrome:
>> Developer Tools >> **Performance** >> ☐ (Record)
...
>> Stop
Same data as CPU Profiler, but in an integrated timeline view

13. Timeline recording



14. Other topics

https://developers.google.com/web/tools/chrome-devtools/rendering-tools/
→ https://developers.google.com/web/tools/chrome-devtools/rendering-tools/js-execution
→ https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/timeline-tool#make-a-recording

Performance Issues and Optimizations in JavaScript: An Empirical Study. Marija Selakovic and Michael Prade. Technical Report. TU Darmstadt, Department of Computer Science. 2015-Oct.
http://mp.binaervarianz.de/JS_perf_study_TR_Oct2015.pdf

Performance Tips for JavaScript in V8 https://www.html5rocks.com/en/tutorials/speed/v8/

https://developers.google.com/speed/articles/optimizing-javascript

Online tests:

- PageSpeed Insights https://developers.google.com/speed/pagespeed/insights/
- Mobile-Friendly Test https://search.google.com/search-console/mobile-friendly

15. Lower level languages

- ASM.js - http://asmjs.org/
- WebAssembly - http://webassembly.org/

16. Exercise: Context promotion

contextPromotion.html

```
<script>
```

```
var dummy;

function foo() {
  var sum = 0;
  for (var i = 0; i < 100000000; i++) {
    sum += Math.sqrt(i);
  }

  setTimeout(function () { dummy = sum; }, 1000);
}

foo(); // This method is too slow and puts too much load on the memory
</script>
Done.
```

Version 0.2
Last updated 2017-05-15 01:31:05 CEST

Memory (garbage collector)

Arnauld Van Muysewinkel
<avm@pendragon.be>
version 0.2, 14-May-2017 Draft version

Table of Contents

1. Content
2. Memory leaks
3. Memory hindrance
4. Leak cause: Global variables
5. Leak cause: Global variables
6. Leak cause: Global variables
7. Leak cause: Global variables
8. Leak cause: Timers or Callbacks
9. Leak cause: Timers or Callbacks
10. Leak cause: Detached nodes
11. Leak cause: Detached nodes
12. Leak cause: Closures
13. Detection techniques
14. Technique: Timeline recording
15. Technique: Timeline recording
16. Technique: Heap snapshot
17. Technique: Heap snapshot
18. Technique: Heap snapshot
19. Technique: Allocation timeline
20. Technique: Allocation timeline
21. Technique: Allocation profiler
22. Technique: Allocation profiler
23. Technique of the 3 snapshots
24. Technique of the 3 snapshots
25. Technique of the 3 snapshots
26. Technique of the 3 snapshots
27. Exercise: Wrong scope
28. Exercise: Detached nodes
29. Exercise: Closure leak
30. Exercise: More...
31. References

1. Content

- Memory leaks
- Detection techniques
- Exercises

(*back to plan*)

2. Memory leaks

- memory leak = some objects are kept in memory, even though not used anymore
- due to some hanging reference *somewhere*

Most frequent causes:

- global variables
- timers or callbacks
- detached nodes (out of DOM references)
- closures

3. Memory hindrance

Even without leak, the memory may be the origin of performance issues:

- too high allocation rate
- hiccups in animation due to GC
- ...

4. Leak cause: Global variables

- global variable in a web page = member of the `window` object
- won't be collected as long as the page is active
- typically due to wrong usage of variable scope in functions

5. Leak cause: Global variables

Default scope

```
function foo(arg) {
    bar = "this is a hidden global variable";
}
```

is equivalent to:

```
function foo(arg) {
    window.bar = "this is a hidden global variable";
}
```

6. Leak cause: Global variables

Wrong usage of `this`

```
function foo(arg) {
    this.bar = "this is a hidden global variable";
}
```

is also equivalent to:

```
function foo(arg) {
    window.bar = "this is a hidden global variable";
}
```

7. Leak cause: Global variables

Tricky usage of `this`

Given:

```
var car = {
  brand: "Nissan",
  setBrand: function(_brand) { this.brand = _brand; }
};

var setCarBrand = car.setBrand;
```

```
setCarBrand("Toyota");
```

is equivalent to:

```
window.brand = "Toyota"
```

Solution:

```
var setCarBrand = car.setBrand.bind(car);
```

8. Leak cause: Timers or Callbacks

Callbacks and cyclic references

Let's consider:

```
var element = document.getElementById('button');
function onClick(event) { element.innerHtml = 'text'; }
element.addEventListener('click', onClick);
// Do stuff
```

We have a loop of references:
`element` → `'click'` event → `onClick` function → `element`

Some (old) browsers do not collect those cyclic references well.

Solution: remove the callback before disposing the `element`:

```
// Now we are about to remove element
element.removeEventListener('click', onClick);
element.parentNode.removeChild(element);
```

9. Leak cause: Timers or Callbacks

Timers

Let's consider:

```
var someResource = ...
setInterval(function() {
  var element = document.getElementById('button');
  if (element) {
    // Do stuff with element and someResource, eg:
    element.innerHTML = someResource.toHtml();
}, 1000)
```

The references graph is like this:
interval → `setInterval` handler function → `element` and `someResource`

Even if the button (referenced through `element`) is removed, the handler (`setInterval`) won't be collected, since the interval is still active. Hence, `someResource` cannot be collected.

Solution: disable the interval when removing the button.

10. Leak cause: Detached nodes

- DOM nodes stored in data structure
- the reference in the data structure is not removed when the DOM node is removed

For example:

```
var elements = {
    buttonElt: document.getElementById('button'),
    imageElt: document.getElementById('image')
};

function doStuff() {
    elements.imageElt.src = 'http://some.url/image';
    elements.buttonElt.click();
}
```

References graph:
- DOM → `'button'` and `'image'` elements
- `doStuff` function → `elements`
.. → `buttonElt` → `'button'` element
.. → `imageElt` → `'image'` element

Even if we remove the button:

```
document.body.removeChild(document.getElementById('button'));
```

it won't be collected, because there is still a reference chain through the `buttonElt` reference stored in `elements`.

11. Leak cause: Detached nodes

Even worst:

```
var elements = {
    tableCell: document.getElementById('td-3-7')
    // where 'td-3-7' is the id of a <td> element inside a big <table>
};
document.body.removeChild(document.getElementById('tab'));
// where 'tab' is the id of this big <table>
```

Even though only one small reference is held by `elements`, the whole table will stay stuck in memory.

Indeed, before the remove, the reference graph is as follows:
DOM → `'tab'` <table> → ... → `'td-3-7'` <td> element → `'tab'` <table>
`elements` → `tableCell` → `'td-3-7'` <td> element → ...

(since each table element holds a reference to the table itself!)

12. Leak cause: Closures

A *closure* is an anonymous function that capture variables **from parent scopes**.

Because of this property, they can be the cause of memory leaks in very specific cases (see exercise).

More info: https://blog.meteor.com/an-interesting-kind-of-javascript-memory-leak-8b47d2e7f156
http://mrale.ph/blog/2012/09/23/grokking-v8-closures-for-fun.html

13. Detection techniques

- timeline recording
- heap snapshot
- allocation timeline
- allocation profiler

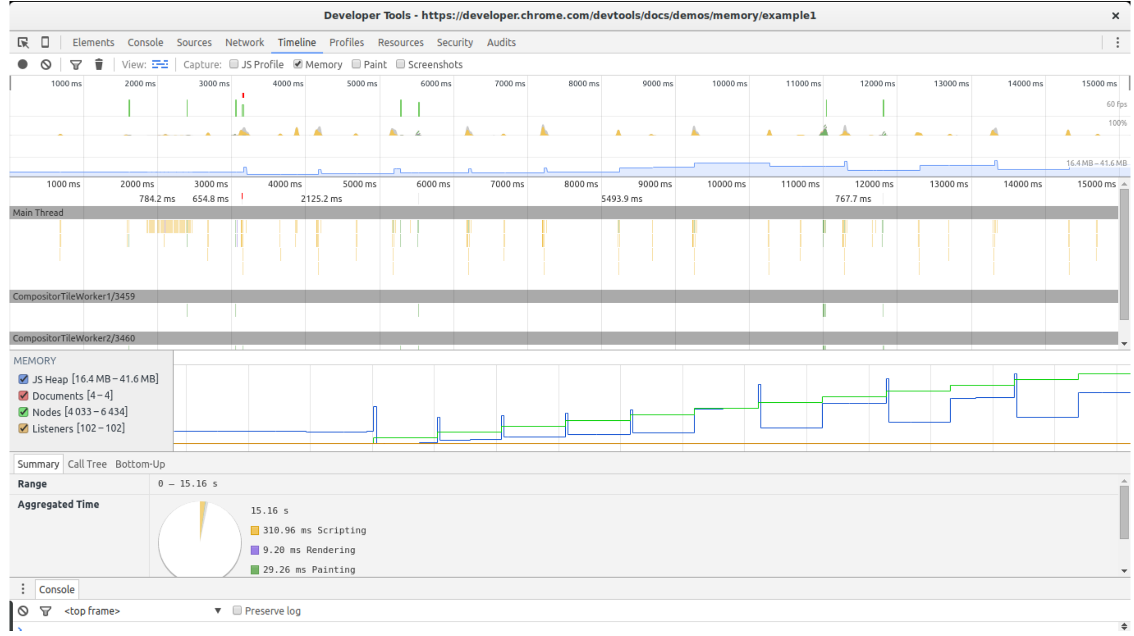
14. Technique: Timeline recording

→ heap size over time

⇒ find out if memory is regularly increasing

in Chrome:
>> Developer Tools >> **Performance >> ⊠ Memory** >> ☐ (Record)

15. Technique: Timeline recording



16. Technique: Heap snapshot

→ capture whole heap

in Chrome:
>> Developer Tools >> Memory >> Profiles >> ⊙ **Take Heap Snapshot** >> Take Snapshot

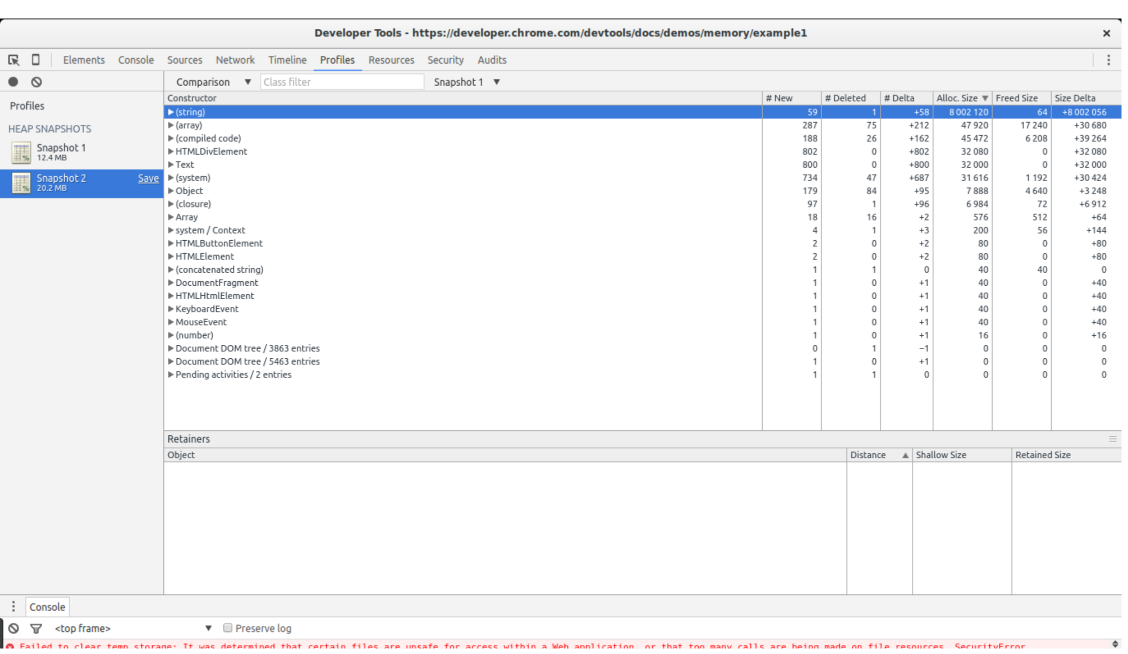
Comparison:

- Take first snapshot immediately after page load
- Take second snapshot after some activity in the page
- Select "Comparison" instead of "Summary"

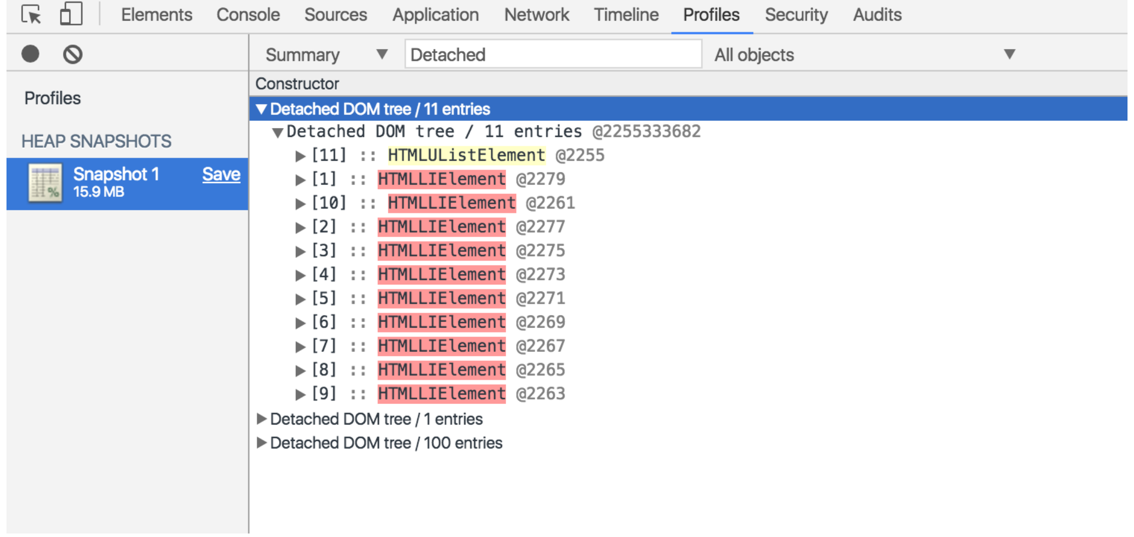
Detached nodes:

- Type "Detached" in the filter box

17. Technique: Heap snapshot



18. Technique: Heap snapshot



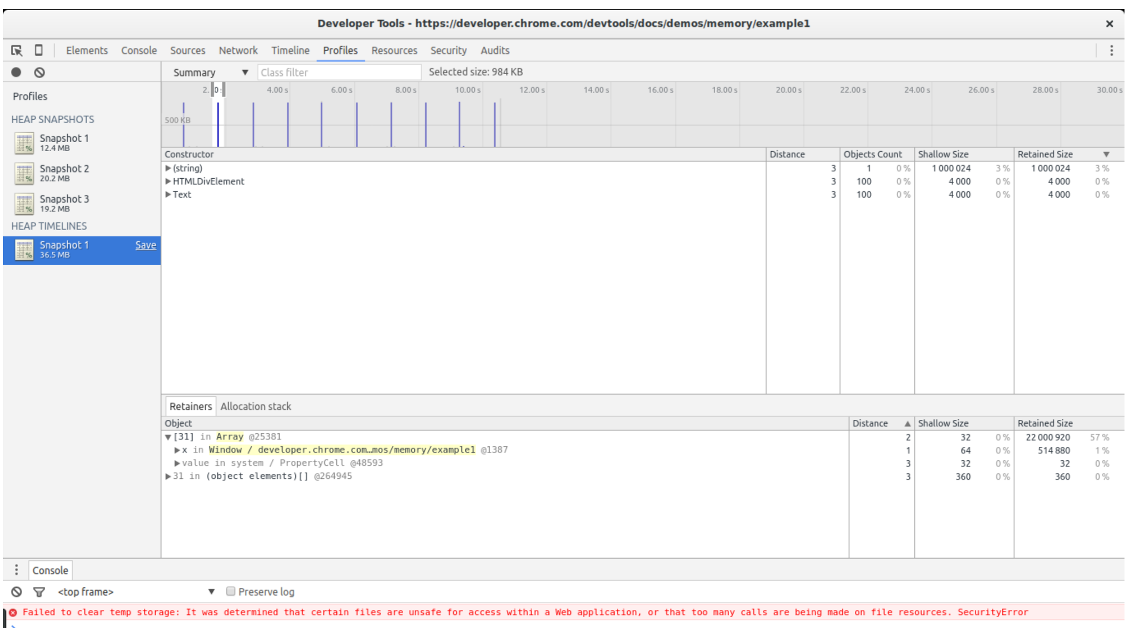
19. Technique: Allocation timeline

→ allocation over time

in Chrome:
>> Developer Tools >> Memory >> Profiles >> ⊙ **Record Allocation Timeline** >> Start

Then select a time interval, to see which allocations occured during the time span.

20. Technique: Allocation timeline



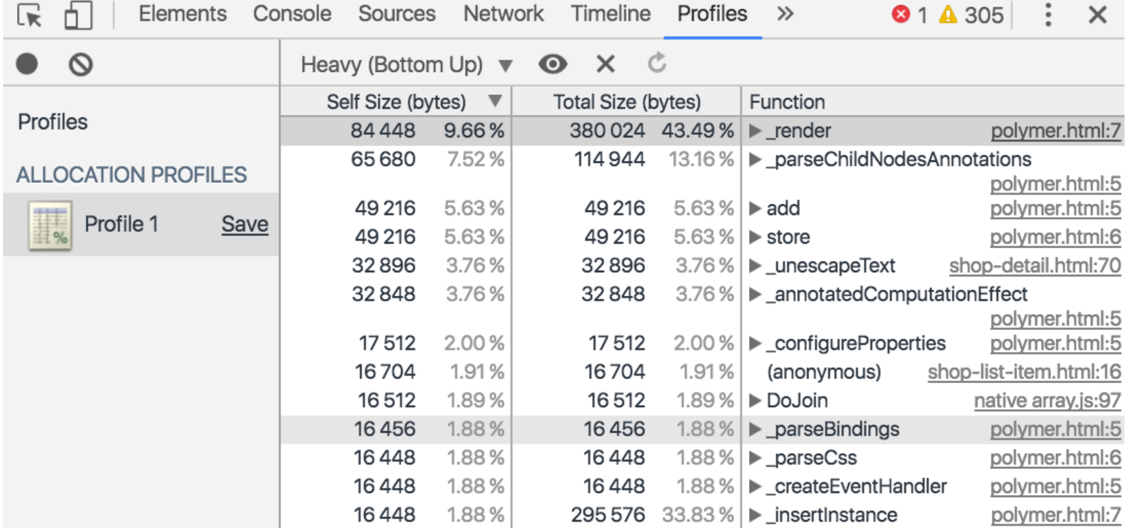
21. Technique: Allocation profiler

→ breakdown of allocations by JavaScript function

in Chrome:
>> Developer Tools >> Memory >> Profiles >> ⊙ **Record Allocation Profile** >> Start

Then, select "Heavy (Bottom Up)" instead of "Chart"

22. Technique: Allocation profiler



Self Size (bytes)		Total Size (bytes)		Function	
84 448	9.66 %	380 024	43.49 %	▶ _render	polymer.html:7
65 680	7.52 %	114 944	13.16 %	▶ _parseChildNodesAnnotations	polymer.html:5
49 216	5.63 %	49 216	5.63 %	▶ add	polymer.html:5
49 216	5.63 %	49 216	5.63 %	▶ store	polymer.html:6
32 896	3.76 %	32 896	3.76 %	▶ _unescapeText	shop-detail.html:70
32 848	3.76 %	32 848	3.76 %	▶ _annotatedComputationEffect	polymer.html:5
17 512	2.00 %	17 512	2.00 %	▶ _configureProperties	polymer.html:5
16 704	1.91 %	16 704	1.91 %	(anonymous)	shop-list-item.html:16
16 512	1.89 %	16 512	1.89 %	▶ DoJoin	native array.js:97
16 456	1.88 %	16 456	1.88 %	▶ _parseBindings	polymer.html:5
16 448	1.88 %	16 448	1.88 %	▶ _parseCss	polymer.html:6
16 448	1.88 %	16 448	1.88 %	▶ _createEventHandler	polymer.html:5
16 448	1.88 %	295 576	33.83 %	▶ _insertInstance	polymer.html:7

23. Technique of the 3 snapshots

→ detect leak remaining after navigating away from a page

Take 3 snapshots as follows:

- Make sure the caches are properly loaded, by navigating in the app.
- Navigate to the page A. Take a snapshot.
- Navigate to the page B. Take a snapshot.
- Navigate to the page A. Take a snapshot.

24. Technique of the 3 snapshots

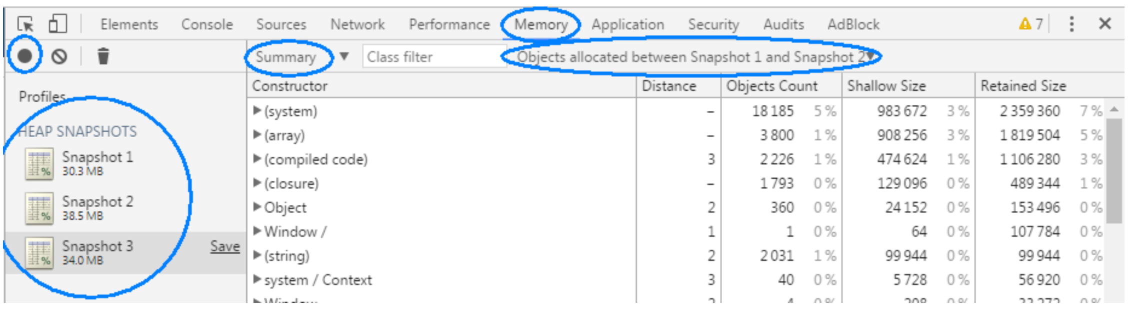
Analyse the data:

- Select snapshot 3
- Select "Summary" mode
- Then, "Objects allocated between Snapshot 1 and Snapshot 2" instead of "All objects"

→ In the snapshot 3 (i.e. only objects existing on the *second* page A), we are filtering by objects that were allocated between the *first* page A and the page B.

In other words, we see only objects from the *first* page A that survived up to the *second* page A.

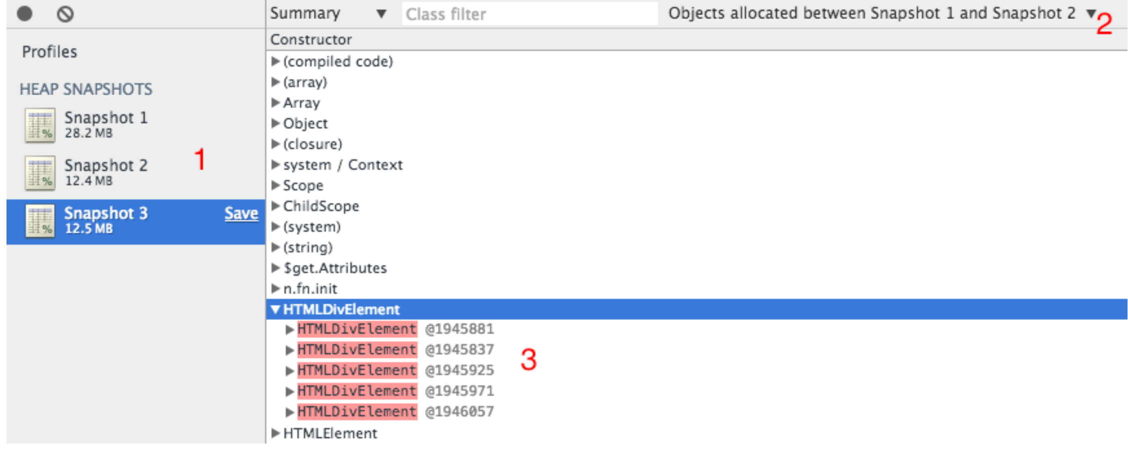
25. Technique of the 3 snapshots



Constructor	Distance	Objects Count		Shallow Size		Retained Size	
▶ (system)	–	18 185	5 %	983 672	3 %	2 359 360	7 %
▶ (array)	–	3 800	1 %	908 256	3 %	1 819 504	5 %
▶ (compiled code)	3	2 226	1 %	474 624	1 %	1 106 280	3 %
▶ (closure)	–	1 793	0 %	129 096	0 %	489 344	1 %
▶ Object	2	360	0 %	24 152	0 %	153 496	0 %
▶ Window /	1	1	0 %	64	0 %	107 784	0 %
▶ (string)	2	2 031	1 %	99 944	0 %	99 944	0 %
▶ system / Context	3	40	0 %	5 728	0 %	56 920	0 %

Figure 1. 3 snapshots technique in Chrome 58.0.3029.110 (64-bit)

26. Technique of the 3 snapshots



27. Exercise: Wrong scope

wrongScope.html

```
<script>
function randChar() {
  return Math.round(Math.random()*36).toString(36)
}
function mkRandString(size) {
  str = new Array(2<<size);
  for (i=0; i<str.length; str[i++] = randChar());
  return str.join('');
}
var data;

data = mkRandString(20);

// Do something...

data = null;
</script>
```

28. Exercise: Detached nodes

detachedDOM.html

```
<button type="button" id="create">Create</button>
<script>
var reg = [];
function mkString(size, fill) {
  return new Array(2<<size).join(fill);
}
function small() { return mkString(8, 'X'); }
function big() { return mkString(20, 'X'); }

function create() {
  var ul = document.createElement('ul');
  for (var i = 10; i-->0;) {
    var li = document.createElement('li');
    li.data = big();
    li.more = small();
    ul.appendChild(li);
  }
  detachedTree = ul;
  reg.push(detachedTree);
}

document.getElementById('create').
    addEventListener('click', create);
</script>
```

29. Exercise: Closure leak

closureLeak.html

```
<script>
var theThing = null;
var replaceThing = function () {
  var originalThing = theThing;
  var unused = function () {
    if (originalThing)
      console.log("hi");
  };
  theThing = {
    longStr: new Array(2<<20).join('*'),
    someMethod: function () {
      console.log(someMessage);
    }
  };
};
setInterval(replaceThing, 5000);
</script>
```

30. Exercise: More…

https://github.com/dwmkerr/angular-memory-leaks

31. References

- Fixing Memory Leaks in AngularJS and other JavaScript Applications. Blog post on dwmkerr.com. http://www.dwmkerr.com/fixing-memory-leaks-in-angularjs-applications/
- http://stackoverflow.com/questions/19621074/finding-javascript-memory-leaks-with-chrome
- BloatBusters: Eliminating memory leaks in Gmail. https://docs.google.com/presentation/d/1wUVmf78gG-ra5aOxvTfYdiLkdGaR9OhXRnOlIcEmu2s/pub
  - > So you found a leak. Now what?
- https://developers.google.com/speed/articles/optimizing-javascript#avoiding-browser-memory-leaks
- Official blog of the V8 JavaScript engine. http://v8project.blogspot.be/
  - Jank Busters http://v8project.blogspot.be/2016/04/jank-busters-part-two-orinoco.html
  - Fall cleaning: Optimizing V8 memory consumption. 2016-10-07. http://v8project.blogspot.be/2016/10/fall-cleaning-optimizing-v8-memory.html

https://developers.google.com/web/tools/chrome-devtools/memory-problems/

https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them/