Server execution of JavaScript: What could possibly go wrong?

Brian Geffon
Staff Software Engineer
Hello!

Brian Geffon
Staff Software Engineer at LinkedIn
San Francisco Bay Area | Computer Software

Current Recommendations
LinkedIn, The Apache Software Foundation
4 people have recommended Brian

Contributor, Apache Traffic Server
The Apache Software Foundation
2011 – Present (3 years)

Committer and PMC member on Apache Traffic Server.

Apache Traffic Server is fast, scalable and extensible HTTP/1.1 compliant caching proxy server. It scales well on modern SMP hardware, handling 10s of thousands of requests per second.
Outline

- Introductions
  - Brief History
    - The paradigm shift
    - Problems!
    - Where we are today
    - Closing thoughts and Questions
LinkedIn in 2003

A single monolithic web application
LinkedIn in 2010

New frameworks: productivity boost.

Data Center

Grails
GSP

JAVA
JSP

Ruby
ERB

HTML

Browser
New Frameworks: Added productivity
Added complexity

- Difficult to maintain numerous versions of the same template

- Make it difficult to share content between apps
Solution: a single templating language

- Do these web app frameworks share anything?
- How can we ensure that we remain D.R.Y.
- What language can be supported across each architecture?
Solution: client side templating

- Web applications return JSON data
- Templates are compiled to JavaScript
- JSON Data is consumed by JavaScript templates which will execute on the client side.
Solution: client side templating, contd.

- Webapps can share UI!

- Ability to cache templates on the client
  - Better performance?
So many options!
The winner: Dust.js

- Dust is a logicless JavaScript templating language
- Dust is extensible
- Dust is inherently D.R.Y.

https://github.com/linkedin/dustjs
Dust.js example

```javascript
(function() {
    dust.register("demo", body_0);

    function body_0(chk, ctx) {
        return chk.write("<h2>").reference(ctx.get("first")).write(" ")
            .reference(ctx.get("last")).write(" is a ").reference(ctx.get("occupation"))
            .write("."), ctx, "h").write("</h2>"),
    }

    return body_0;
})(());
```

gets compiled into a JavaScript function

+ `{"name": "Fizz Bang"} = <h2>Hello Fizz Bang!</h2>`
Outline

- Introductions
- Brief History
- The paradigm shift
  - Problems!
  - Where we are today
  - Closing thoughts and Questions
The paradigm shift

- Reusable UI gives rise to component sharing across apps
- Components are now separated from data models
- Ability to avoid RTT for components embedded in page.
The paradigm shift: Fizzy
What’s going on here?

Web
fz.js

1

Fizzy
Server

2

Profile

4

Contacts

3

Profile
What’s does the application return?

HTTP/1.1 200 OK
Content-Type: text/html
X-FS-Page-Parse: 1
X-FS-Page-Id: profile-view-fs
X-FS-Host-Id: ela4-appxxxx.prod

```html
<html>
<body>
<script type="fs/embed"
  fs-uri="http://www.linkedin.com/profile/summary"
  fs-id="example-id">
</script>
</body>
</html>
```
What do these embedded components return?

HTTP/1.1 200 OK  
Content-Type: application/json  
X-FS-Page-Id: profile-activity  
X-FS-Host-Id: ela4-appxxxx.prod  
X-FS-TL: http://cdn-host/hash-of-template1.js  
X-FS-Template-Keys: __default__=hash-of-template1

{"first": "Brian", "last": "Geffon", "Occupation": "Software Engineer"}
What does the browser see?

```html
<html>
<head>
  <script type="text/javascript" src="/static/fz.js"></script>
</head>
<body>
  <script type="text/javascript"
          src="http://cdn-host/hash-of-template1.js"></script>
  <code id="example-id-content">
    <!-- {"first":"Brian","last":"Geffon", "occupation": "."} -->
  </code>
  <script type="text/javascript">
    fs.embed("example-id", "hash-of-template1");
  </script>
</body>
</html>
```
Yay! A fancy new web architecture

- Components are now stand alone
- Nice UI separation
- Reusability
What could possibly go wrong?

- Large JSON payloads caused many problems with IE7
  - IE7 doesn’t have a native JSON parser!
What could possibly go wrong?

- Some older browsers would take a very long time executing JS
  - Many browsers didn’t have optimized JS engines
What could possibly go wrong?

- Search Engine Optimization
  - JS in GoogleBot Yes, many others: No
Server Side Rendering (SSR)

- Unfortunately we need a way to execute JavaScript on the server

- Could potential performance improvements been seen across the board?
The Pieces of SSR

- High performance caching HTTP proxy

- High performance embeddable JavaScript Engine

Google V8 JS Engine
Server Side Rendering: What’s going on here?
What’s does the application return?

HTTP/1.1 200 OK
Content-Type: text/html
X-FS-Page-Parse: 1
X-FS-Page-Id: profile-view-fs
X-FS-Host-Id: ela4-appxxxx.prod

```html
<html>
<body>
<script type="fs/embed"
    fs-uri="http://www.linkedin.com/profile/activity"
    fs-id="example-id"
    fs-render-control="server">
</script>
</body>
</html>
```
What does the browser see?

```html
<html>
<head>
  <script type="text/javascript" src="/static/fz.js"></script>
</head>
<body>
  <h2>Brian Geffon is a Software Engineer.</h2>
</body>
</html>
```
Yay! A fancy new web architecture

- We can now support old web browsers
- We can now gracefully handle SEO
- It turns out that even for modern browsers sometimes we can execute JavaScript faster!
Outline

- Introductions
- Brief History
- The paradigm shift
- **Problems**
  - Where we are today
  - Closing thoughts and Questions
What could possibly go wrong?

- A shared JS engine gives rise to issues and vulnerabilities that don’t affect browsers that execute JS.
What could possibly go wrong?

- **Context Pollution**
  - One malicious request can poison the context of another
  - This issue exists with any dynamic language
Context Pollution

Silly example but illustrates the need for isolation.

What if we leave off `var` in JavaScript?
Context Pollution: The solution

- Each request requires its own context
  - Completely reload the environment and bootstrap code

- Performance Hits?
What could possibly go wrong?

- Poorly written JavaScript can take forever to execute!
Although this is tail recursion and a silly example, it illustrates the need for stack protection and time limitations.
Long Running JavaScript: The solution

- Enforce stack size limits that allow you to gracefully kill a VM

- Sandbox: accept that apps will misbehave and allow them to only hurt themselves.
Long Running JavaScript: The solution

- Execution limits (we use 1000ms)

- Exponentially decay the execution limit to prevent taking down the entire site!
What could possibly go wrong?

- Garbage Collection!
Garbage Collection!

Queue times going through the roof!
The culprit: Garbage Collection!
GC tuning: it takes practice.

Avg Queue times < 0.3ms, P99.99 < 2ms.
GC Tuning

- Adjust old generation to be several order of magnitudes less than new generation

- New generation is critical because of the short lived jobs and contexts.

- More Threads!
Ideas for the future

- User load times are actually improved with SSR: do it 100% of the time.

- A generic JS engine: allow apps to return any JavaScript, not just Dust.js
Questions?