There and Back Again: Server Side JavaScript

Douglas Crockford
Yahoo!

The Web was intended to be a document retrieval system.

Early on people felt the need for something more dynamic.

CGI and Perl scripts.

Templating

- PHP [1995] demonstrated an easier model, where script blocks could be inserted into an HTML file. Each time the page is requested, the scripts would run, injecting text into the HTML stream.
- PHP was highly influential: ASP and JSP.
- Netscape LiveWire [1996]. Used server side JavaScript in a PHP-like way.
- Fortunately, it failed.

It was fortunate that it failed because there are problems with the HTML template approach.

- Security: It is too easy to inject dangerous text into the HTML stream, enabling XSS Attacks. It is possible to inject text correctly, but it is hard.
- Performance: Scripts are run serially and are blocking. If the page contains many independent components, unnecessary delays are imposed. It is possible to do things in parallel, but it is hard.
- No one wants to do hard in PHP. That’s not what it’s for.

PHP made a lot of sense 15 years ago.

A lot has changed in 15 years.

JavaScript is very successful in the browser with an event-driven, non-blocking model.

Can we bring that success back to the server?

Obvious Advantages

- Web developers only been to be current in one language.
- Using Ajax techniques to build HTML components is much safer than templating because structures are guaranteed to be encoded correctly.
- Complex pages can be built up concurrently instead of serially.

Obvious Disadvantage

- Outside of web development, the completely event-driven model is still unfamiliar.
- Historically, programming languages going back to FORTRAN relied on blocking I/O.
- In the 1970s, researchers and game developers were experimenting with event driven systems.

1984

Archivio Composizione Esposizione Strumenti
The Macintosh™ Finder
Bruce Horn and Steve Capps
512K
Version 4.1 ©1985 Apple Computer

You have to write the programs inside out! Waa!

Let’s go back to the command line.

Home

Welcome to HyperCard

HyperCard is a unique software tool that allows you to do more with your computer.

With HyperCard, you use documents called **stacks**. Stacks can help you do many different things—for example, you could use a stack to keep track of your appointments, manage your expenses, learn a new language, or record and play sounds.

To learn more, click “What is HyperCard?,” to the right.

©1993-1995 Apple Computer, Inc. All Rights Reserved.

Home

What is HyperCard?

Addresses

Audio Help

Non-programmers were incredibly productive with HyperCard.

Suddenly, professional programmers got smarter and there was an explosion of Mac and Windows applications.

HyperCard was all about events

- Programs (aka stacks) were written as a collection of event handlers attached to visible objects.
- Events bubble up.

```
on mouseUp

on keyDown

on cardEnter

on idle
```

HyperCard had a big impact on the evolution of the browser.

JavaScript is well suited for this model.

As awful as the DOM is,
JavaScript+DOM is effective.

JavaScript+YUI3 is really effective.

JavaScript does not have **`READ`**.

That has always been seen as a huge disadvantage, but it is actually a wonderful thing.

`READ` is blocking, and blocking is bad for event loops.

JavaScript programmers are smarter about using event loops than programmers of other languages.

Event loop is just one approach to concurrency.

The most popular approach is threading:

Two or more real or virtually CPUs sharing the same memory.

Threading

Pro

- No rethinking is necessary.
- Threading is compatible with blocking I/O.
- Execution continues as long as any thread is not blocked.

Con

- Stack memory per thread.
- If two threads use the same memory, a race *may* occur.
- To be continued…

Two threads

1. `my_array[my_array.length] = 'a';`
2. `my_array[my_array.length] = 'b';`

- `['a', 'b']`
- `['b', 'a']`

Two threads

1. `my_array[my_array.length] = 'a';`
2. `my_array[my_array.length] = 'b';`

- `['a', 'b']`
- `['b', 'a']`
- `['a']`
- `['b']`

```
my_array[my_array.length] = 'a';
```

```
length_a = my_array.length;
my_array[length_a] = 'a';
if (length_a >= my_array.length) {
    my_array.length = length_a + 1;
}
```

```
my_array[my_array.length] = 'a';
```

```
length_a = my_array.length;
length_b = my_array.length;
my_array[length_a] = 'a';
if (length_a >= my_array.length) {
my_array[length_b] = 'b';
    my_array.length = length_a + 1;
}
if (length_b >= my_array.length) {
    my_array.length = length_b + 1;
}
```

It is impossible to have application integrity when subject to race conditions.

Mutual Exclusion

- semaphore
- monitor
- rendezvous
- synchronization

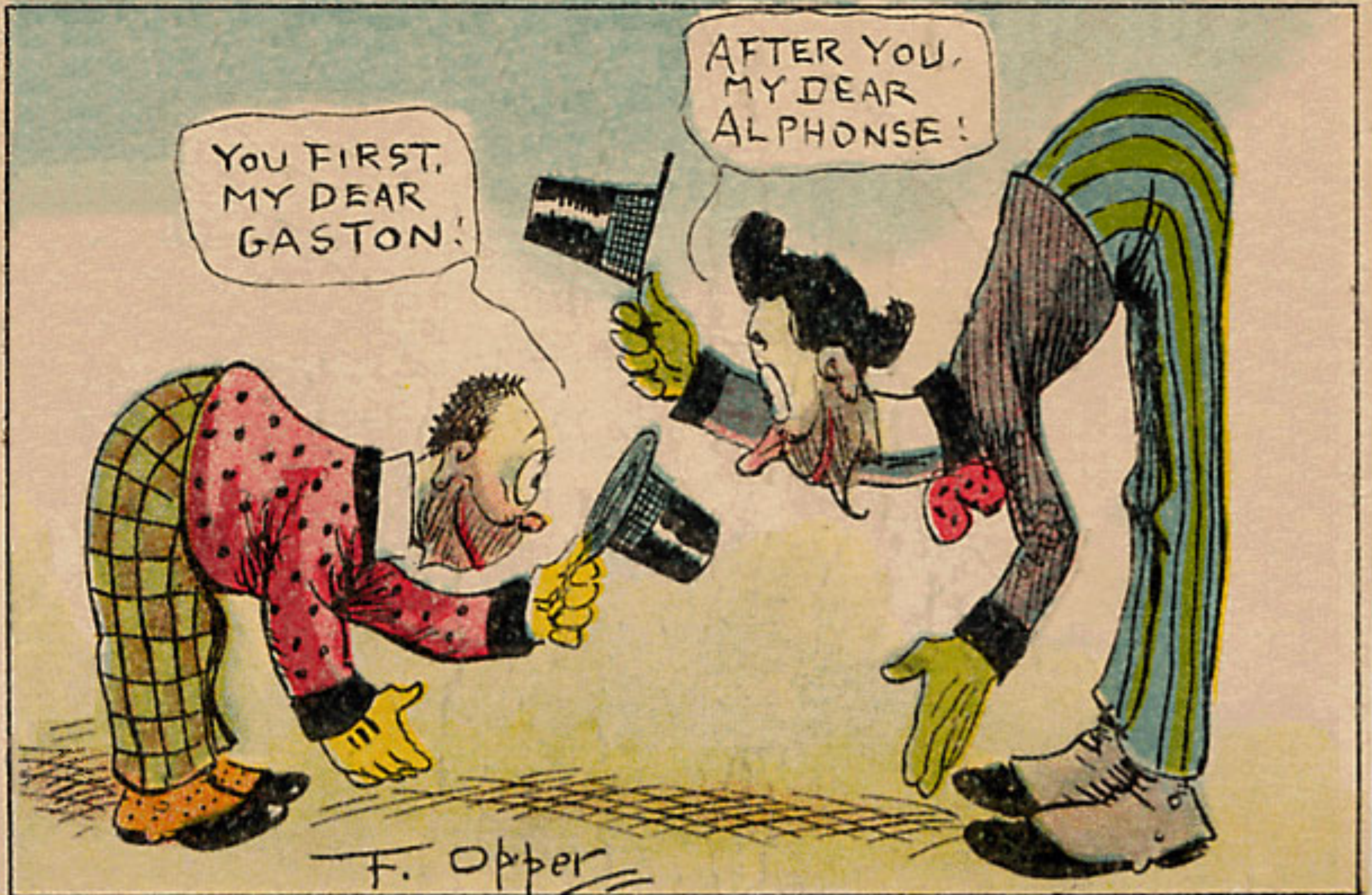
- This used to be operating system stuff.
- It has leaked into applications because of networking and the multi-core problem.

Mutual Exclusion

- Only one thread can be executing on a critical section at a time.
- All other threads wanting to execute the critical section are blocked.

- If threads don’t interact, then the program runs at full speed.
- If they do interact, then races will occur unless mutual exclusion is employed.
- Mutual exclusion can cause threads to block.

Deadlock



COPYRIGHT, 1906, BY AMERICAN JOURNAL EXAMINER.
"YOU FIRST, MY DEAR."

Deadlock

Deadlock

- Deadlock occurs when threads are waiting on each other.
- Races and deadlocks are difficult to reason about.
- They are the most difficult problems to identify, debug and correct.
- They are often unobservable during testing.
- Managing sequential logic is hard. Managing temporal logic is really, really hard.

Threading

Pro

- No rethinking is necessary.
- Blocking programs are ok.

Con

- Stack memory per thread.
- If two threads use the same memory, a race *may* occur.
- Overhead.
- Deadlock.
- Thinking about reliability is extremely difficult.
- System/Application confusion.

Fortunately, there is a model that completely avoids all of the reliability hazards of threads.

The Event Loop!

Event Loop

Pro

- Completely free of races and deadlocks.
- Only one stack.
- Very low overhead.
- Resilient. If a turn fails, the program can still go on.

Con

- Programs must never block.
- Programs are inside out! Waa!
- Turns must finish quickly.

Long running tasks

- Two solutions for long running programs:
- Eteration: Break the task into multiple turns.
- Move the task into a separate process (workers).

Remote Procedure Call

- Combines two great ideas, functions and networking, producing a really bad idea.
- Like **`READ`**, attempts to isolate programs from time. The program blacks out.
- In reading the program, it is by design difficult to see where time is lost.
- This can result in a terrible experience for the user. Lost time === annoying delays.
- Keeping the user waiting without warning is disrespectful and rude.

Latency Compensation

- At a minimum, acknowledge user’s input immediately.
- Don’t lock up the interaction while waiting for the server’s response.
- In some applications, it is reasonable to predict the server’s response and display it immediately. Later display a correction if the prediction was wrong.

Security

XS

S

XSS has two causes:

1. Sharing of the global object.

2. Misinterpretation of HTML...

What can an attacker do if he gets some script into your page?

An attacker can request additional scripts from any server in the world.

Once it gets a foothold, it can obtain all of the scripts it needs.

An attacker can read the document.

The attacker can see everything the user sees.

An attacker can make requests of your server.

Your server cannot detect that the request did not originate with your application.

If your server accepts SQL queries, then the attacker gets access to your database.

SQL was optimized for SQL Injection Attacks

An attacker has control over the display and can request information from the user.

The user cannot detect that the request did not originate with your application.

An attacker can send information to servers anywhere in the world.

The consequences of a successful attack are horrible.

Harm to customers.

Loss of trust.

Legal liabilities.

The browser does not prevent any of these terrible things.

Web standards require these weaknesses.

15 Years of XSS

Tragically, HTML5 ignores and worsens the XSS problem.

The browser is a loaded gun pointed at your head.

This pulls the trigger:

```
<?= "bang" ?>
```

Page Templates

- The page template systems (PHP, ASP, JSP…) are not a good match for the way we build modern sites.
- A template is too rigid a framework.
- It is too easy to insert text into a context where it can be misinterpreted and executed, completing an XSS attack.

Can we do better by using JavaScript on the server?

There are some obvious advantages:

We can take advantage of our new understanding of JavaScript.

What about Server Side JavaScript with an Event Loop?

node.js

- node.js implements a web server in a JavaScript event loop.
- It is a high-performance event pump.

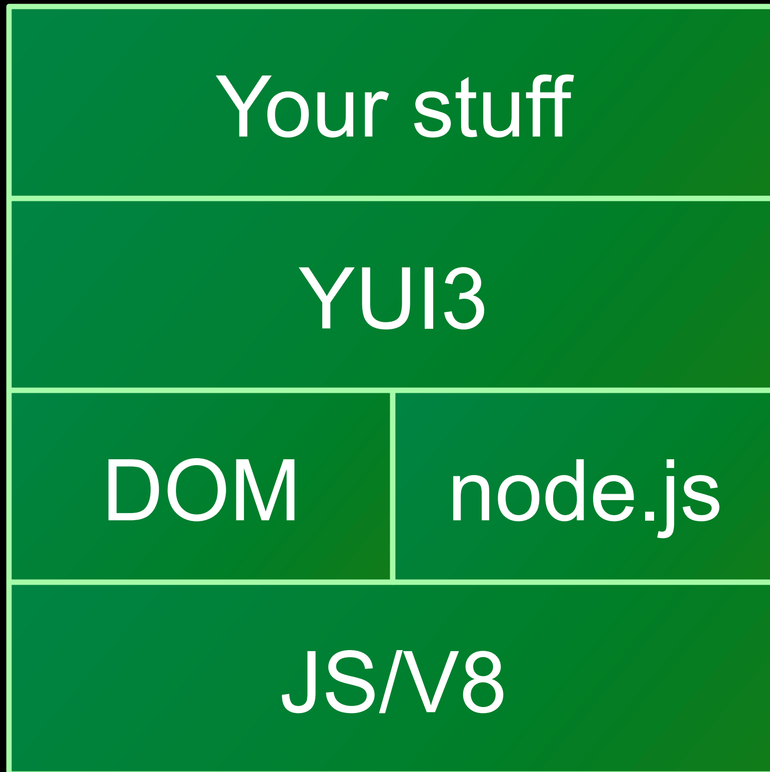
`fs.read(`fd, length, position, encoding,
  `function` (err, str, bytesRead) **`{...})`**

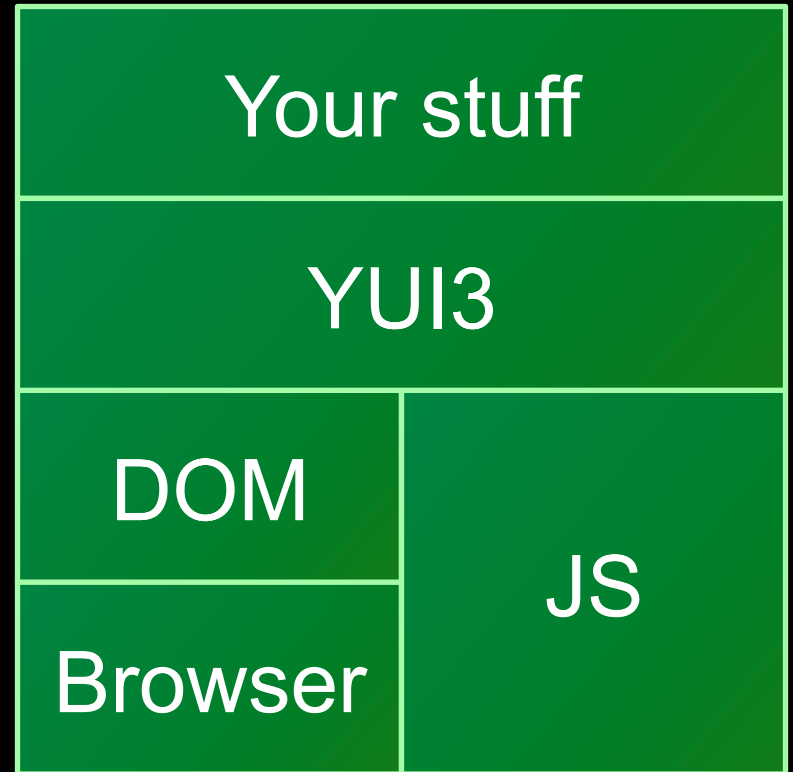
- Everything is (or can be) non-blocking.
- Except:

    some synchronous functions

    `require`

Your stuff runs on both sides





Exceptions

- Exceptions do not work in an event system, because exceptions only work in the current turn. An exception cannot be caught by a previous turn.
- So APIs need to have callbacks in pairs: One callback for the successful case, and one callback for the exceptional case.

Deeply nest callback functions.

Research into better patterns and library support.

Requestor

```
myRequestor = function (sync) {
    service_request(arguments,
            function (result) {
        sync(result, error);
    });
};

parallel([requestors…], sync, timeout);
serial([requestors…], sync, timeout);
```