CS 188

Scalable Internet Services

John Rothfels

November 5, 2019

Today's Agenda

The Client-side Renaissance

- **Better JavaScript**
  - Client-side JavaScript grows up
  - Overview of ES6
- **Post-JavaScript**
  - Asm.js
  - Emscripten

Motivation

- http://backbonejs.org/examples/todos/
- http://coolwanglu.github.io/vim.js/experimental/vim.html
- https://developer.mozilla.org/en-US/demos/detail/bananabread

The Browser Wars

- Mid-90s Netscape reigns supreme
- Microsoft releases initial version of Internet Explorer in 1995
- Competition between Netscape and Microsoft produces significant innovation in browsers
  - JavaScript
  - Cookies
  - CSS

The Browser Wars

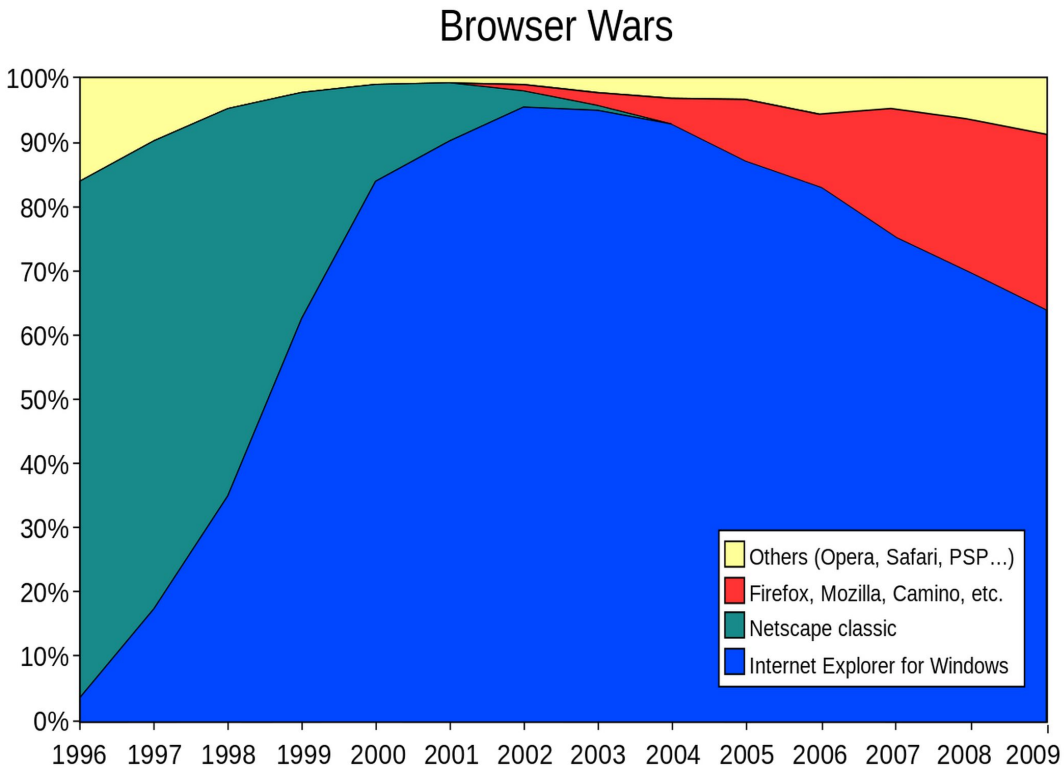
Microsoft bundles Internet Explorer to Windows 98

- Every file management window is a browser
- Eventually triggers an antitrust lawsuit against Microsoft

Meanwhile, Netscape focuses on open-sourcing its browser

- Eventually creates the Mozilla foundation
- Acquisition by AOL

The Browser Wars



Microsoft wins. Internet Explorer becomes the dominant browser for roughly a decade.

The Browser Wars

Version	Release Year
IE 1	1995
IE 2	1995
IE 3	1996
IE 4	1997
IE 5	1999
IE 6	2001
IE 7	2006
IE 8	2009

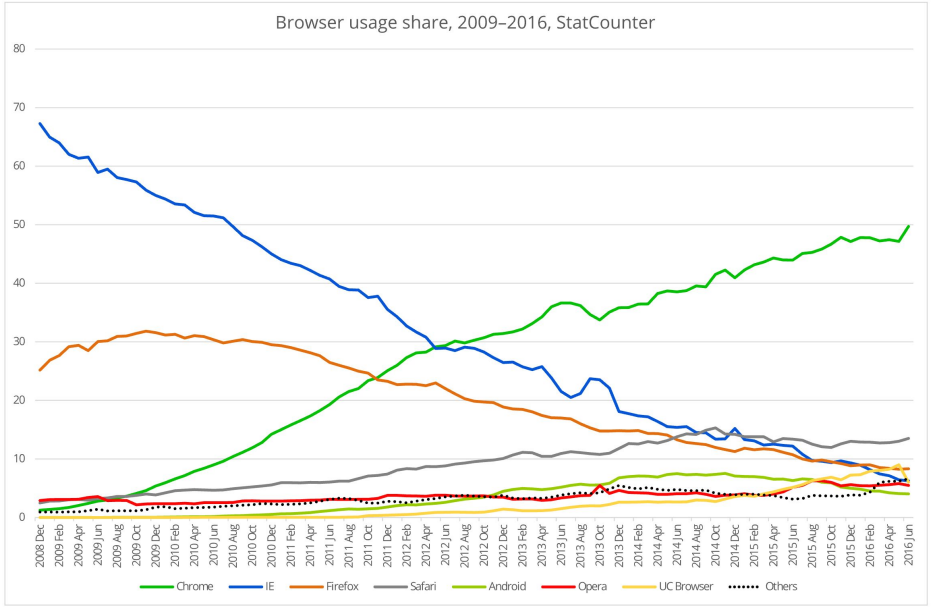
Lull in browser innovation commences

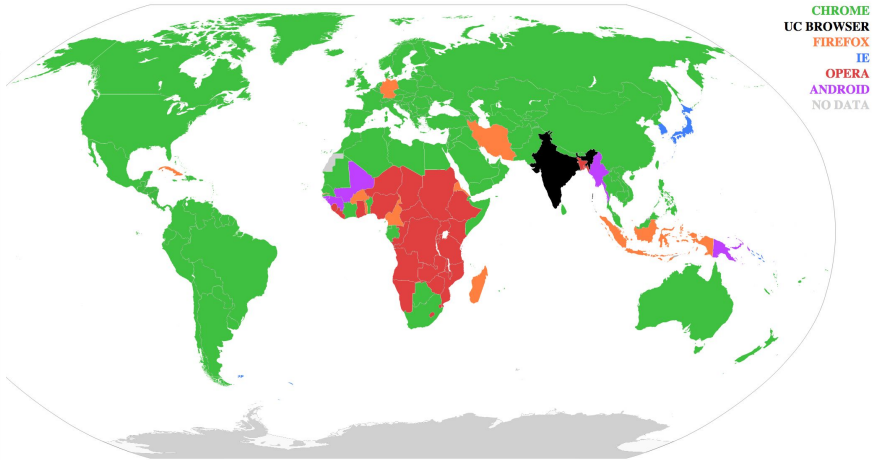
- Lack of competition means no reason to innovate
- Time between releases increases significantly

The Browser Wars

- Due to a variety of factors, Microsoft slowly loses market share to Firefox (Mozilla)
  - Security
  - Performance
- As Microsoft is slowly bleeding market share, Google announces Chrome
- Browser innovation reignites, today there are at least 4 viable browsers.

Browser Market Share Today





Source: http://en.wikipedia.org/wiki/Usage_share_of_web_browsers

Client-side renaissance

During the browser dark ages, three things eventually spur the client side renaissance

- XMLHTTPRequest
- DOM Manipulation
- V8

XMLHttpRequest (Ajax)

Allows JavaScript on the page to asynchronously request resources from the server.

```
var req = new XMLHttpRequest();
req.onload = function() {
  console.log(“I’m a callback!”);
};
req.open("get", "/comments", true);
req.send();
```

Originally added to IE to enable the Outlook Web Access team to asynchronously communicate with the server (year ~2000).

- Other browsers implement it, becomes de facto standard by ~2004.

XMLHttpRequest (Ajax)

Prior to this, two-way communication with the server meant a full page refresh

- Either a clicked link or a submitted form.

This method allows javascript in your browser to send requests and receive responses completely programmatically.

- Example: As you type, google.com will show you intermediate results.

Most people don't use this method directly. jQuery is commonly used as an abstraction layer:

```
$.get('/comments', function(data) { alert("I'm a callback!");});
```

XMLHttpRequest (Ajax)

Security limitation:

- These requests can only go to the originating domain that the Javascript was served from.
- This prevents randomsite.com from (say) accessing your gmail inbox
  - All requests use existing cookies, of which your session is one.
- This works pretty well for things like Google Analytics, they can phone home to Google, but can't access the server-side resources of the website they are on.

This technique is now referred to as Ajax

- Asynchronous JavaScript and XML
- XML was originally envisioned as the transport format, but there is nothing XML-specific about it.
- Today, XML is not as regularly used as JSON for transport.

DOM Manipulation

- Document Object Model
  - A standardized way of representing the structure of a web page as a tree of in-memory objects
  - These objects are accessed via Javascript and can be queried and manipulated
- Example:

```
var newDiv = document.createElement("div");
var newContent = document.createTextNode("Hello World!");
newDiv.appendChild(newContent);
document.body.appendChild(newDiv);
```

DOM Manipulation

Progression of DOM Manipulation:

- DOM Level 0: Navigator 2, IE 3 (~1995).
  - Allowed reading the values of forms and links.
  - Not standardized (called legacy DOM)
- DOM Level 1: Navigator 4, IE 5 (~1998)
  - Allowed access and modification of anything by index
    - `document.forms[1].elements[2]`
- DOM Level 2: ~ 2000
  - added getElementById(), DOM event model
- DOM Level 3: ~2004, current
  - XPath support, keyboard event handling

V8: Javascript Gets Fast

September 2008, Google releases Chrome

- In addition to other novel features, it includes the V8 Javascript engine
- V8 applies modern, state of the art VM techniques to Javascript
  - Not interpreted: dynamically compiled to machine code
  - Garbage collector is fast
    - Generational: separates allocated memory into young and old groups, treats them differently
    - Incremental: doesn't need to perform all GC at once
  - Applies other modern VM optimizations:
    - Inlining, code elision, inline caching, etc.

V8: Javascript Gets Fast

V8 treats Javascript performance seriously, and triggers other browsers to do the same.

- Safari's JavaScriptCore
- IE's Chakra
- Firefox's SpiderMonkey

Today these VMs are all roughly evenly matched.

- Performance leader goes back and forth, but in the same ballpark

Sidenote: V8 was designed to also work well outside of the browser. It is the execution engine that Node.JS is built on.

Client-side Renaissance

So, by 2008, we have all the ingredients ready for a client-side renaissance:

- Globally installed virtual machines
- ...that can present content that can communicate via Ajax to the internet service they originated
- ...that have full programmatic control of the user interface (DOM).
- ...that use modern, high performance VM techniques
- … that exist in a competitive marketplace
  - Four viable browsers, available on multiple OSes
  - Competing to stay ahead of the pack
  - Standards-compliance is a competitive advantage

These are things we enjoy today that we did not always have.

Client-side Renaissance

What do these modern client-side applications look like?

- Instead of being a series of pages requested from a web server, we can serve up a running javascript application
  - This application is regularly sending user input back to the server
  - This application is regularly receiving structured data instead of rendered markup.
- These applications generally persist through user interactions
  - Clicks don't necessarily mean full-page refreshes
- Communication with the server is decoupled from user interaction
  - While the browser sits open, a javascript timer can go and check for new data and update the page as needed

Client-side Renaissance

Consequences of this shift

- Client side logic is much more complex and full page refreshes are more rare
- It's possible to build applications that work "offline"
- It's possible to build effective "push" mechanisms
- The "running application" is much more static and cacheable
- The apis you build to serve up structured data can be used by mobile applications and other internet services
- "Real" Javascript VMS enable very ambitious use of CPU resources

Client-side Renaissance

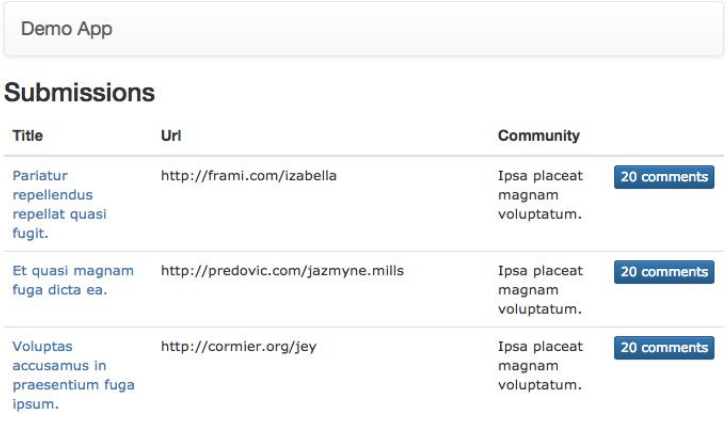
So you’re interested in building a web application that moves much of its logic and rendering to the client…

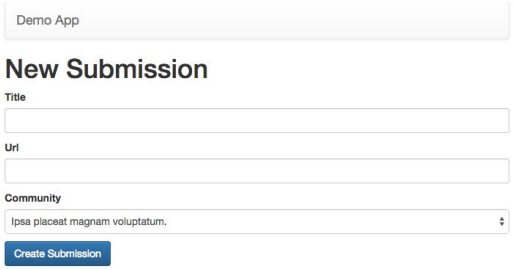
Let’s look at an example of this transition, in our Demo app.

Client-side Renaissance

In a traditional web application:

- When you click the “New Submission” button, the browser makes a new HTTP request and loads the response
- The response is an entire web page, and with it are numerous assets.
- The page returned has form elements
- When you fill out the form and submit it, the server may find it invalid and send back another form.
- List of submissions only changes when you refresh the page.

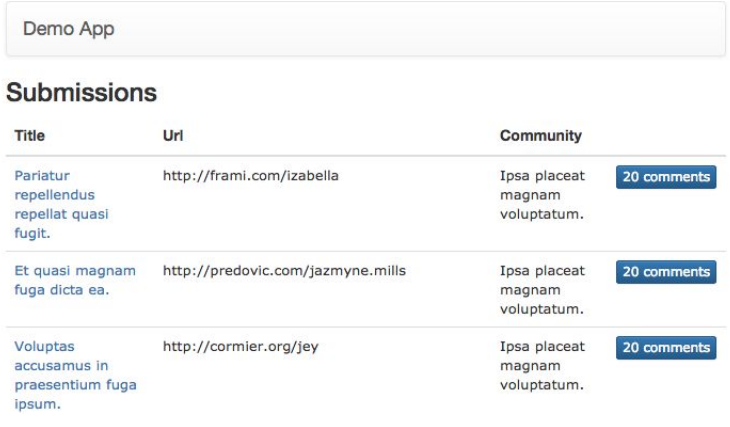


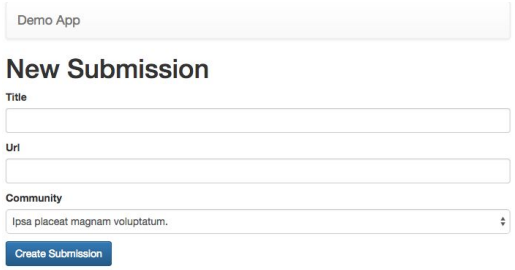


Client-side Renaissance

In a client-side web application:

- When you click the “New Submission” button, javascript executes and redraws the page to show a form. No HTTP requests occur.
- When you fill out the form and submit it, the input is validated in the browser using javascript.
- If valid, an Ajax request is sent to the server
- List of submissions only changes live.





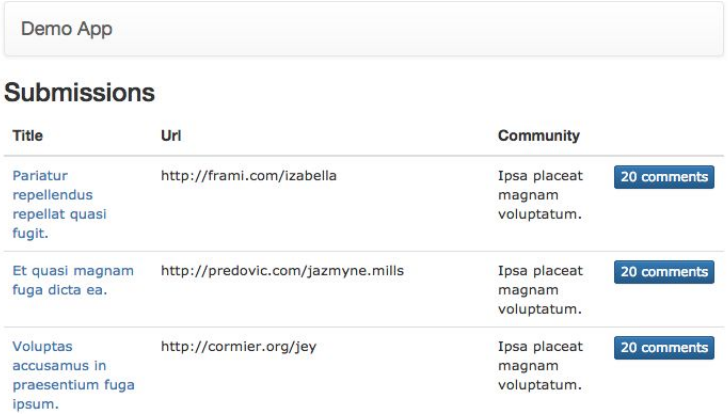
Client-side Renaissance

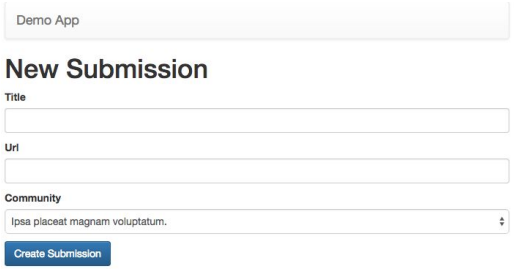
Benefits:

- UI is extremely responsive
- Less network traffic
- Live updates!

Costs:

- Client side code is much more complex.





Client-side Renaissance

Client side code is much more complex

- Before, the client was mostly displaying a static page.
- Now, the client must:
  - Understand the relationship between input events and corresponding DOM updates
  - Understand enough application logic to distinguish valid input from invalid input
  - Keep a persistent connection to the server and display updates as they come in.

Client-side Renaissance

How should our application design adjust to this drastic increase in complexity?

- One design approach: “A tangled pile of jQuery selectors and callbacks, all trying frantically to keep data in sync between the HTML UI, your JavaScript logic, and the database on your server.”

Client-side MVC

How should our application design adjust to this drastic increase in complexity?

- ~~One design approach: "A tangled pile of jQuery selectors and callbacks, all trying frantically to keep data in sync between the HTML UI, your JavaScript logic, and the database on your server."~~
- One popular approach to managing this complexity is the use of MVC frameworks on the client.

Client-side MVC

MVC: Model-View-Controller

- You all know it from Rails
- Separates the presentation (View) of your data from the data itself (Model).
- Controller exists to accept and coordinate updates to the Models.
- Models encapsulate business logic and state.

There are many client side frameworks that implement variations of these. Today we will talk about two:

- Angular
- React

Client-side MVC

Angular.js

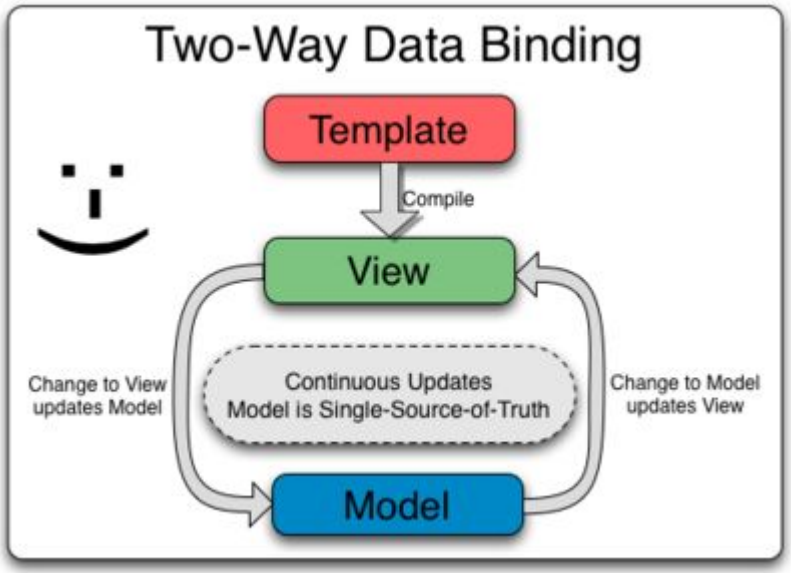
- MVC framework supported and promoted by Google.
- Large and complex
- Suitable for Single Page Applications
- Emphasis on declarative style for building UI
- Uses two-way data binding

Notable websites using Angular:

- AWS console, HBO, VirginAmerica

Client-side MVC

Two-way data binding:



Client-side MVC

Data Binding Example:

```
<div ng-app ng-init="qty=1;cost=2">
  <div>
  Quantity: <input type="number" min="0" ng-model="qty">
  </div>
  <div>
    Costs: <input type="number" min="0" ng-model="cost">
  </div>
  <div><b>Total:</b> {{qty * cost | currency}}</div>
</div>
```

http://plnkr.co/edit/EpVlAulGMdHymMakqGMx?p=preview

Client-side MVC

Model Example:

```
<div ng-controller="Controller">
  Hello <input ng-model='name'> <hr/>
  <span ng-bind="name"></span> <br/>
</div>

angular.module('docsBindExample', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.name = 'Andrew';
}]);
```

Client-side MVC

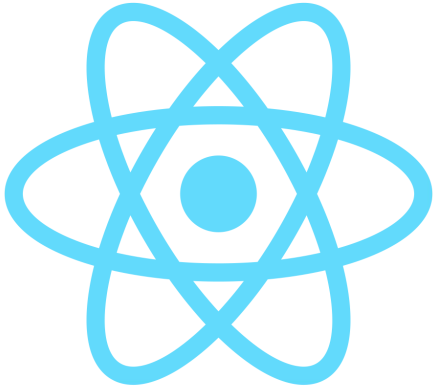
Angular Highlights:

- Ambitious and large framework that really turns the page into an application
- Data binding provides a lot of magic
- A framework you adopt wholesale
- Declarative style retains HTML traditional nature

React

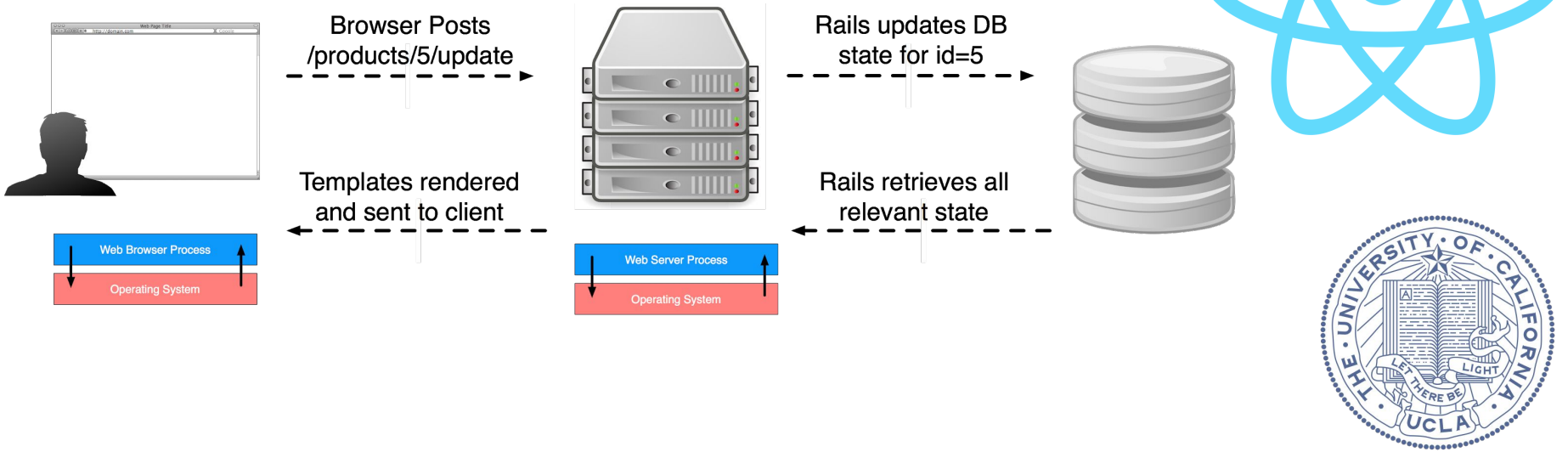
React

- Developed at Facebook
- Observation: Event handling systems can be hard to reason about
  - Two-way binding helps take care of some of this automatically, but it can still be complex



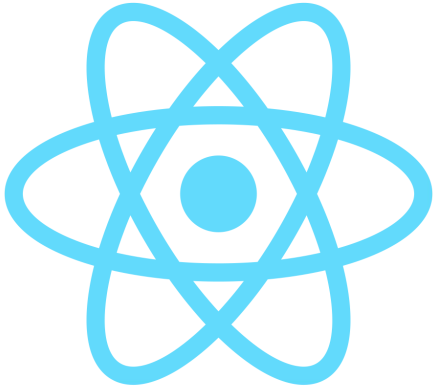
Why are server-side rendered systems simple to reason about?

React


Browser Posts
/products/5/update
Rails updates DB
state for id=5
Templates rendered
and sent to client
Rails retrieves all
relevant state
Web Browser Process
Operating System
Web Server Process
Operating System

React

If we could completely re-render the entire UI whenever anything changes, we could develop systems that were very simple to reason about.



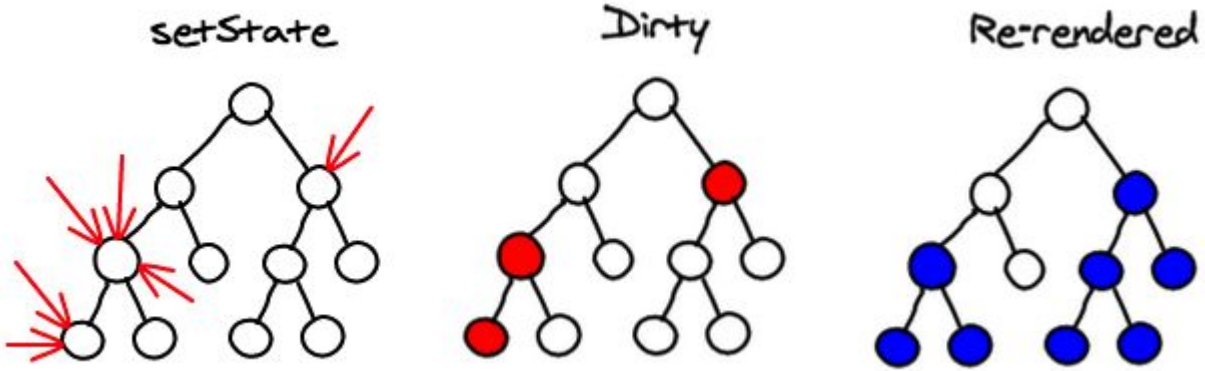
We can’t do this because it would be too slow.

React allows you to build simple-to-reason-about UI code by making these operations fast.

React

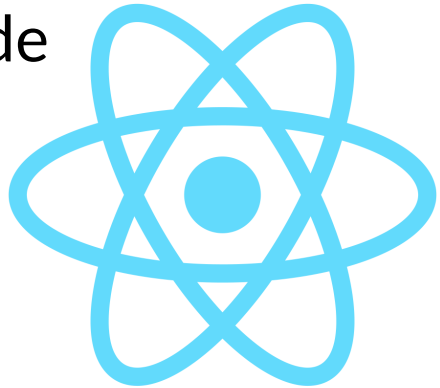
VirtualDOM

- Instead of updating the DOM directly, keep track of modifications in a “VirtualDOM”, and only re-render what is needed.



React

By giving the software engineer the ability to code as though they are re-rendering everything each time, we get simple UI code:

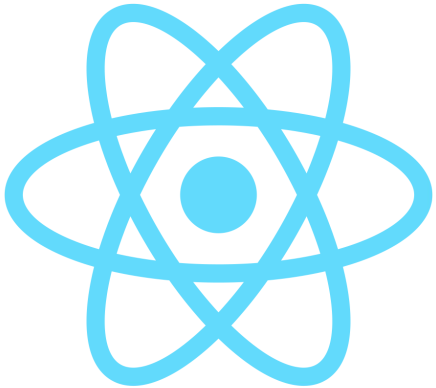


```
function HelloMessage(props) {
  return <div>Hello {props.name}</div>
}

React.render(<HelloMessage name=”Scalable Internet Systems” />, mountNode);
```

React

Currently in production use at Facebook, Instagram and Khan Academy, and many more.



Easy to adopt incrementally.

React-native

- Build native mobile apps using these same techniques
- Not significant code-reuse from web to mobile, but toolchain reuse.
- Supports iOS and Android

Overview of ES6

ECMAScript 6: New version of JavaScript

- Node.js and richer clients mean increasing use of JavaScript
- As usage of the language expands, ES6 represents a push to improve it.
- Current browser support is limited, but transpilers exist to use it today
  - Traceur, Babel

Overview of ES6

Block-scoped variables

```
// Function scope (var)
function order(x, y) {
  if (x > y) {
    var tmp = x;
    x = y;
    y = tmp;
  }
  console.log(tmp===x);
    // true

  return [x, y];
}
```

```
// Block scope (let,const)
function order(x, y) {
  if (x > y) {
    let tmp = x;
    x = y;
    y = tmp;
  }
  console.log(tmp===x);
    // ReferenceError:
    // tmp is not defined
  return [x, y];
}
```

From Axel Rauschmayer, Aaron Frost, Fluent 2015: http://cdn.oreillystatic.com/en/assets/1/event/125/ECMAScript%206%20in%20theory%20and%20practice%20Presentation.pdf

Overview of ES6

Destructuring

Extract multiple values via patterns:

```
let obj = { first: 'Jane', last: 'Doe' };
let { first: f, last: l } = obj;
    // f='Jane', l='Doe'
```

Can be used for:

- variable declarations (`var`, `let`, `const`)
- assignments
- parameter definitions

From Axel Rauschmayer, Aaron Frost, Fluent 2015: http://cdn.oreillystatic.com/en/assets/1/event/125/ECMAScript%206%20in%20theory%20and%20practice%20Presentation.pdf

Overview of ES6

Destructuring: arrays

```
let [x, y] = ['a', 'b'];
    // x='a', y='b'

let [x, y, ...rest] = ['a', 'b', 'c', 'd'];
    // x='a', y='b', rest = [ 'c', 'd' ]

[x,y] = [y,x];  // swap values

let [all, year, month, day] =
    /^(\d\d\d\d)-(\d\d)-(\d\d)$/
    .exec('2999-12-31');
```

From Axel Rauschmayer, Aaron Frost, Fluent 2015: http://cdn.oreillystatic.com/en/assets/1/event/125/ECMAScript%206%20in%20theory%20and%20practice%20Presentation.pdf

Overview of ES6

Multiple return values

```
function findElement(arr, predicate) {
    for (let index=0; index < arr.length; index++) {
        let element = arr[index];
        if (predicate(element)) {
            return { element, index };
                    // same as { element: element, index: index }
        }
    }
    return { element: undefined, index: -1 };
}
let a = [7, 8, 6];

let {element, index} = findElement(a, x => x % 2 === 0);
    // element = 8, index = 1
let {index, element} = findElement(···); // order doesn't matter

let {element} = findElement(···);
let {index} = findElement(···);
```

From Axel Rauschmayer, Aaron Frost, Fluent 2015: http://cdn.oreillystatic.com/en/assets/1/event/125/ECMAScript%206%20in%20theory%20and%20practice%20Presentation.pdf

Overview of ES6

Modules: named exports

```
// lib/math.js
let notExported = 'abc';
export function square(x) {
    return x * x;
}
export const MY_CONSTANT = 123;
```

```
// main1.js
import {square} from 'lib/math';
console.log(square(3));
```

```
// main2.js
import * as math from 'lib/math';
console.log(math.square(3));
```

From Axel Rauschmayer, Aaron Frost, Fluent 2015: http://cdn.oreillystatic.com/en/assets/1/event/125/ECMAScript%206%20in%20theory%20and%20practice%20Presentation.pdf

Overview of ES6

Modules: default exports

```
//----- myFunc.js -----
export default function (...) { ... }

//----- main1.js -----
import myFunc from 'myFunc';
```

```
//----- MyClass.js -----
export default class { ... }

//----- main2.js -----
import MyClass from 'MyClass';
```

From Axel Rauschmayer, Aaron Frost, Fluent 2015: http://cdn.oreillystatic.com/en/assets/1/event/125/ECMAScript%206%20in%20theory%20and%20practice%20Presentation.pdf

Overview of ES6

Method definitions

```
let obj = {
    myMethod() {
        ...
    }
};

// Equivalent:
var obj = {
    myMethod: function () {
        ...
    }
};
```

From Axel Rauschmayer, Aaron Frost, Fluent 2015: http://cdn.oreillystatic.com/en/assets/1/event/125/ECMAScript%206%20in%20theory%20and%20practice%20Presentation.pdf

Overview of ES6

Parameter default values

Use a default if a parameter is missing.

```
function func1(x, y='default') {
    return [x,y];
}
```

Interaction:

```
> func1(1, 2)
[1, 2]
> func1()
[undefined, 'default']
```

From Axel Rauschmayer, Aaron Frost, Fluent 2015: http://cdn.oreillystatic.com/en/assets/1/event/125/ECMAScript%206%20in%20theory%20and%20practice%20Presentation.pdf

Overview of ES2015

Rest parameters

Put trailing parameters in an array.

```
function func2(arg0, ...others) {
    return others;
}
```

Interaction:

```
> func2('a', 'b', 'c')
['b', 'c']
> func2()
[]
```

No need for `arguments`, anymore.

From Axel Rauschmayer, Aaron Frost, Fluent 2015: http://cdn.oreillystatic.com/en/assets/1/event/125/ECMAScript%206%20in%20theory%20and%20practice%20Presentation.pdf

Overview of ES6

Spread operator (...): function arguments

```
Math.max(...[7, 4, 11]); // 11

let arr1 = ['a', 'b'];
let arr2 = ['c', 'd'];
arr1.push(...arr2);
    // arr1 is now ['a', 'b', 'c', 'd']

// Also works in constructors!
new Date(...[1912, 11, 24]) // Christmas Eve 1912
```

Turn an array into function/method arguments:

- The inverse of rest parameters
- Mostly replaces `Function.prototype.apply()`

From Axel Rauschmayer, Aaron Frost, Fluent 2015: http://cdn.oreillystatic.com/en/assets/1/event/125/ECMAScript%206%20in%20theory%20and%20practice%20Presentation.pdf

Overview of ES6

Arrow functions: less to type

```
let arr = [1, 2, 3];
let squ;

squ = arr.map(function (a) {return a * a});
squ = arr.map(a => a * a);
```

From Axel Rauschmayer, Aaron Frost, Fluent 2015: http://cdn.oreillystatic.com/en/assets/1/event/125/ECMAScript%206%20in%20theory%20and%20practice%20Presentation.pdf

More safety: static types

As JavaScript applications have became larger and more common, pressure increased for better developer tooling:

- automated refactors
- go-to-def, find refs
- type safety, null safety

Today many projects use TypeScript (from Microsoft) or Flow (from Facebook) to improve dev velocity and code maintainability. They compile to JavaScript and are at least as expressive as ES6.

Polyglot Browser

Javascript-based web applications have taken over many domains

- Email, calendaring, word processing, social, etc.
- HTML5 has Geo-location, WebRTC, etc.

What can’t web browsers do?

Polyglot Browser

Web browsers struggle with compute heavy tasks...

- Games
- Speech recognition
- Image recognition
- Image processing, effects

Lots of work has been put into making JS fast, but its still a scripting language.

Polyglot Browser

Competitive pressure on browser vendors.
More and more is desired from browsers.

Javascript engine performance has been improving

- Dynamic compilation
- Advanced garbage collection
- Other VM optimizations

Polyglot Browser

There is no other language on the client.

- `<script type=”text/c”>` doesn’t exist
- This means we are limited in the compute performance on the client.
- This also means we don’t have language heterogeneity on the client
  - Why is this bad?

Polyglot Browser

Key observation:

- Javascript engines are limited in the performance they can deliver because of the rich feature set of Javascript.
  - Garbage collection, Dynamic dispatch, etc.
- If Javascript didn't have some of these features, we could build really high performance VMs
- **More usefully, if restrict ourselves to the fast subset of features, VM designers can make that go very fast.**

Asm.js

Example:

```
f = function(a, b){
  return a + b;
}
f(5, 2)
=> 7

f(“foo”, “bar”)
=> “foobar”
```

Asm.js

Example:

```
f = function(a, b){
  return a + b;
}
f(5, 2)
=> 7

f(“foo”, “bar”)
=> “foobar”
```

How is `f()` compiled to machine code?

Which “+” should be used?

Depends on the args passed to `f()` at runtime

Asm.js

Example:

```
f = function(a, b){
  a = a|0;
  b = b|0;
  return a + b;
}
f(5, 2)
=> 7
```

The bitwise `|0` doesn't affect the value, but guarantees the result is an integer

We've removed the need to check types to implement "+"

Asm.js

Another example:

```
result = ""
for(i=10; i>0; i--){
  result += "a"
}
```

In this code, the variable “result” is getting repeatedly reassigned.

The garbage collector here is cleaning up after each.

Asm.js

Another example:

```
result = ""
for(i=10; i>0; i--){
  result += "a"
}
```

If strings are immutable in Javascript, how do we avoid this?

Asm.js

Another example:

```
buff = Int32Array(10);
for(i=10; i>0; i--){
  buff[i] = "a".charCodeAt(0)
}
result =
String.fromCharCode.apply(
  String, buff);
```

Avoid the garbage collector completely.

Allocate a reusable array of integers and manage your memory by hand.

Asm.js

A subset of Javascript that VMs can execute very fast.

All valid JS. Any interpreter can execute it, but some will optimize.

Developed at Mozilla in 2013

Asm.js

Only number types are used.

- No strings, booleans, or objects
- Higher level concepts can be built on numbers (like traditional assembly language)

Asm.js

No VM-provided GC

- All non-stack data is stored in one large array, called the heap.
- It is up to the application code to manage this heap (like traditional assembly language)

Asm.js

No dynamic dispatch

- By only using numbers, many dispatch issues are avoided completely
- Coercion is applied to ensure compiler can be sure about types:

```
var x = a | 0
var y = + b
```

Asm.js

Asm.js is organized into modules. Modules are a series of function definitions with references to the JS stdlib, a foreign function interface, and a heap array for storage

```
function MyAsmModule(stdlib, foreign, heap) {
    "use asm";  // marks this function as an asm.js module
    // module body:
    function f1(...) { ... }
    function f2(...) { ... }
    ...
    return {
        export1: f1,
        export2: f2,
        ...
    };
}
```

Asm.js

The string “use asm”; is a hint to the VM that the code should be asm-compliant.

```
function DiagModule(stdlib) {
    "use asm";
    var sqrt = stdlib.Math.sqrt;
    function square(x) {
        x = +x;
        return +(x*x);
    }
    function diag(x, y) {
        x = +x;  y = +y;
        return +sqrt(square(x) + square(y));
    }
    return { diag: diag };}
```

Asm.js

The string “use asm”; is a hint to the VM that the code should be asm-compliant.

If FFI or heap is not passed in, defaults are provided.

```
// Browsers: this === window
var fast = DiagModule(this);

console.log(fast.diag(3, 4));  // 5
```

Asm.js

So, by restricting our use of Javascript to static typing and manual memory management, we can generate JS that can be executed very quickly.

Who would ever want to write code like this?

LLVM

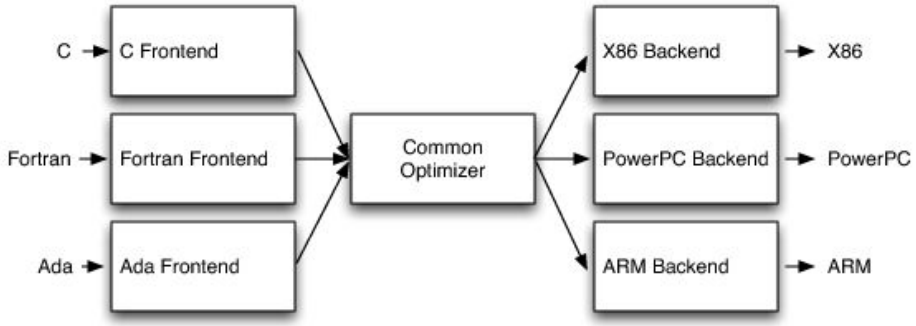
LLVM: Low Level Virtual Machine project

- Language-agnostic low level bytecode that can be translated to machine code.
- Many languages support generating LLVM: C, C++, Python, Ruby, Rust, Scala, etc.

LLVM

Basic idea:

- Have many front-end libraries able to generate LLVM bytecode
- LLVM optimizations are language/architecture agnostic
- Many back-ends to generate bytecode for each architecture.



LLVM

Sample:

```
#include<stdio.h>

int main() {
  printf("hello, world!\n");
  return 0;
}
```

LLVM

Sample:

```
@.str = private unnamed_addr constant [15 x i8] c"hello, world!\0A\00", align 1
; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1
  %2 = call i32 (i8*, ...)* @printf(
    i8* getelementptr inbounds ([15 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}
```

Asm.js

So we have many languages compiling to LLVM, and many backends that translate that to machine instructions.

What do we get if we write a backend that translates to Javascript?

Asm.js

If you have used the clang compiler, you are using LLVM.

Pyston, Rubinius, Safari Webkit all use it.

Emscripten

Emscripten translates LLVM bytecode into Asm.js

Emscripten

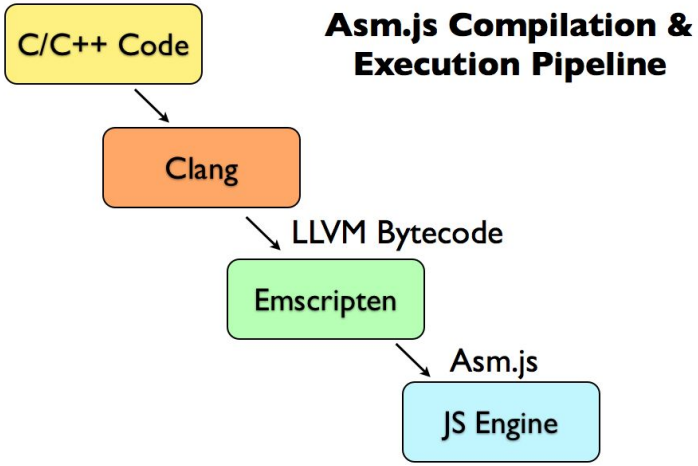
Example:

```
//LLVMIR
define i32 @func(i32* %p){
  %r= load i32* %p
  %s= shl i32 %r, 16
  %t= call i32 @calc(i32 %r, i32 %s)
  ret i32 %t
}
```

```
// JS
function func(p){
  var r = HEAP[p];
  return calc(r, r << 16);
}
```

Emscripten

Because we have many languages that compile to LLVM, we can now use Emscripten to execute these languages in browsers



Emscripten

What happens to application performance?

- What % performance decrease do you think?

Emscripten

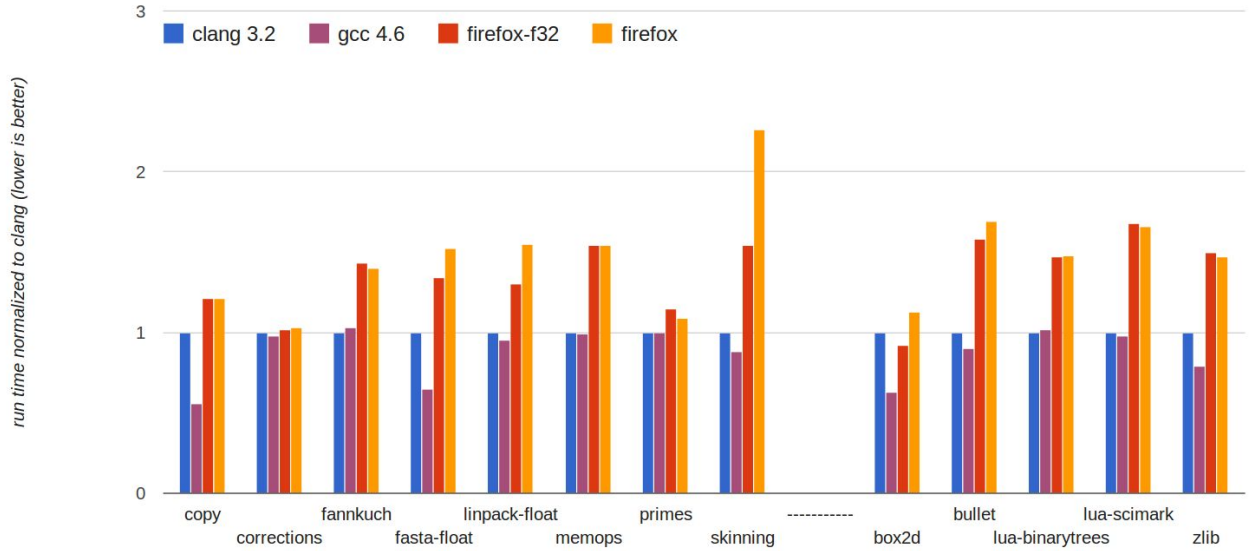
What happens to application performance?

- What % performance decrease do you think?

Project authors suggest 2X slowdown over native code!

Emscripten

What happens to application performance?



MRuby

Demo!

http://joshnuss.github.io/mruby-web-irb/

http://pypyjs.org/

MRuby

Why don’t we see client-side frameworks being built in these languages?

MRuby

Why don’t we see client-side frameworks being built in these languages?

- jquery: 96KB
- angular: 441KB
- MRuby: 1.7MB
- PyPyJS: 2.7MB

Conclusion

What does this mean in the long term?

There is a path towards language heterogeneity on the client.

Javascript won't necessarily take over all of web application development.

For Next Time...

Focus on finishing features and load testing your application.

Over halfway through quarter so reach out for help if you get stuck.