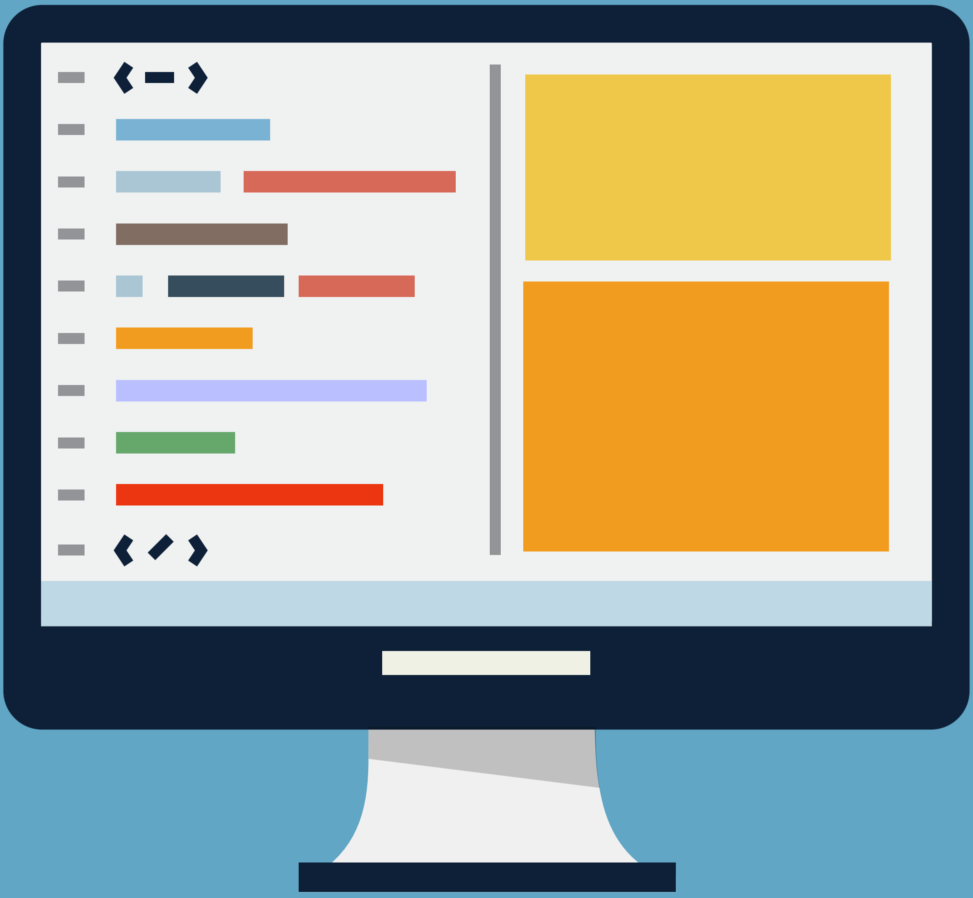
FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

NEXT GENERATION HTML5 AND JAVASCRIPT

eMag Issue 25 - March 2015



InfoQ

ARTICLE
The Modern JavaScript Developer's Toolbox

Q&A
Matthew Carver on The Responsive Web

ARTICLE
React in Real Life at Codecademy

The Modern JavaScript Developer's Toolbox

The toolbox of the modern JavaScript developer has changed a lot in the last 20 years. From IDEs to build automation tools, there are plenty of options for developers. Choosing the best JavaScript library is only part of the process. Linters, bundlers, and minifiers are essential to productivity and performance when building modern web apps.

Virtual Panel: Real-World JavaScript MVC Frameworks

JavaScript front-end codebases grow larger and more difficult to maintain. As a way to solve this issue developers have been turning to MVC frameworks which promise increased productivity and maintainable code. InfoQ asked the opinion of experts practitioners about how they use these frameworks and the best practices they follow when developing JavaScript applications.

NEXT GENERATION HTML5 AND JAVASCRIPT

InfoQ

ARTICLE
The Modern JavaScript Developer's Toolbox

Q&A
Matthew Carver on The Responsive Web

ARTICLE
React in Real Life at Codecademy

Q&A with Matthew Carver on The Responsive Web

Responsive web design is an important part of the modern web and a must-have skill for web developers. The Responsive Web by Matt Carver provides an introduction and suggestions on how to get started. Beginning with the what and why and moving all the way to some advanced techniques, Carver provides a solid overview of an essential tool in the modern web developer's toolbox.

React in Real Life at Codecademy

Codecademy recently switched to React.js for their front-end learning environment. While many React examples are basic, author Bonnie Eisenman goes over how to use React in a large, critical environment.

Towards a Resolution-Independent Web with SVG

This article examines the advantages of using Scalable Vector Graphics (SVG) as the preferred format for the graphic assets of any web or mobile web project. The aim here is not to deter designers and developers from the proven method of using raster formats (PNG / JPEG) but rather to showcase how usage of SVGs can enhance the workflow of any web project.

QCon NEW YORK

Jun 8-12, 2015

Conference for Professional Software Developers
qconnewyork.com

SAVE $100 WHEN YOU REGISTER WITH PROMO CODE "JAVASCRIPT"

Tracks for QCon NY 2015

- **Applied Data Science and Machine Learning** - Putting your data to use. The latest production methods for deriving novel insights
- **Engineering Culture** - Building and scaling a compelling engineering culture
- **Modern Advances in Java Technology** - Tips, techniques and technologies at the cutting edge of modern Java
- **Monoliths to Microservices** - How to evolve beyond a monolithic system -- successful migration and implementation stories
- **The Art of Software Design** - Software Arch as a craft, scenario based examples and general guidance
- **Architecting for Failure** - War stories and lessons learned from building highly robust and resilient systems
- **Emerging Technologies in Front-end Development** - The state of the art in client-side web development
- **Fraud Detection and Hack Prevention** - Businesses are built around trust in systems and data. Securing systems and fighting fraud throughout the data in them.
- **High Performance Streaming Data** - Scalable architectures and high-performance frameworks for immediate data over persistent connections
- **Reactive Architecture Tactics** - The how of the Reactive movement: Release It! techniques, Rx, Failure Concepts, Throughput, Availability
- **Architectures You've Always Wondered about** - Learn from the architectures powering some of the most popular applications and sites
- **Continuously Deploying Containers in Production** - Production ready patterns for growing containerization in your environment
- **Mobile and IoT at Scale** - Trends and technologies for building mobile applications
- **Modern Computer Science in the Real World** - How modern CS tackles problems in the real world
- **Optimizing Yourself** - Maximizing your impact as an engineer, as a leader, and as a person

DAVID IFFLAND is InfoQ's HTML5/JavaScript Lead Editor. He is the founder of Heavy Code and Principal Software Developer at the University of Illinois. He tweets at @daveiffland.

A LETTER FROM THE EDITOR

JavaScript is chaotic. The pace of change is faster than ever and it seems like a new framework or important library pops up every couple of weeks. A major shift in the language is about to hit when ECMAScript 6 is finalized this year.

Modern web developers have to juggle more constraints than ever before; standing still is not an option. We must drive forward and this eMag is a guide to getting on the right track. We'll hear from developers in the trenches building some of the most advanced web applications and how to apply what they've learned to our own work.

David Haney of StackOverflow starts us off with a description of what's in the toolbox of a JavaScript developer in 2015. IDEs are just the start. Gone are the days of dropping jQuery into a project as the first and only JavaScript library. Whether it's AngularJS, ReactJS, or something else that's popped up in the last ten minutes, choosing the right JavaScript libraries is an important step. By adding gulp, linters, bundlers, and minifiers to the toolchain, the modern developer can deliver high-quality, fast JavaScript and CSS – perfect for the modern web.

InfoQ editor Dio Synodinos hosts a virtual panel with leaders of the JavaScript community to talk about MVC frameworks. They cover the kinds of problems frameworks solve, mobile, performance, and workflows. The article provides a glimpse behind the keyboard of JavaScript developers around the globe.

Much of this eMag covers JavaScript, but as long as it's client-side, all the JavaScript in the world won't matter without HTML5. We're expected to build web apps and sites that work not just on a desktop browser, but on a mountain of mobile devices. I have the pleasure of interviewing Matt Carver, author of The Responsive Web, to hear what he thinks about the challenges of building responsive web sites.

In 2010, Apple debuted the iPhone 4 and the first "Retina Display", a term used to describe displays with high pixel densities. While this was great for users, designers and developers now have to contend with images that don't look right on all screens. Angelos Chaidas guides us through the use of SVG to combat this excessive work. Beyond simple uses, Angelos not only covers adding animation and interactivity, but also provides insights on how SVG fits into the workflow.

Wrapping up, Bonnie Eisenman tells the story of how Codecademy started using React.js in production. Created by Facebook, React.js boasts one of the fastest browser rendering systems, and it's starting to catch the attention of developers everywhere. Online, so many samples and demos are "toy" apps that don't get to the meat of what a production app needs. Bonnie shows us what works and what doesn't along with guidance for introducing React to your web app.

HTML5 and JavaScript are the oldest parts of the web, yet they're changing every day. I hope this eMag will help you come up to speed on the current state of web development and assist you on your quest for glory.

The Modern JavaScript Developer's Toolbox

David Haney is the core-team engineering manager at Stack Exchange, creator of question and answer websites such as Stack Overflow and Server Fault. He spends his days supporting developers by solving problems and improving processes. He was previously a lead developer on Fanatics' e-commerce platform, which hosts over 10,000 websites including the NFL Shop (official online shop of the NFL) and NBAStore.com (the official NBA store). David is the creator of Dache, an open-source distributed-caching framework. His spare time is spent drinking beer and participating in community user groups and programming events, often simultaneously.

JavaScript is a scripting language initially designed to enhance web pages but is now used in almost every way imaginable. Advances have been made that allow JavaScript to run on the server side as well as to be compiled into native phone application code. Today's JavaScript developer is part of a rich ecosystem filled with hundreds of IDEs, tools, and frameworks. With so many options and resources, some developers may find it difficult to know where to begin.

A quick jaunt through history

Let's take a quick trip back to 1995 when Netscape Navigator and Internet Explorer 1.0 were the browsers of choice. Websites had annoying blinking text and far too many GIFs. A site full of rich content could take two minutes to load on a dial-up connection. Along the way, a web language was born that allowed these ancient websites to execute client-side code. This was the year that gave birth to JavaScript.

The websites of 20 years ago didn't use JavaScript much, and certainly not to its full potential. Occasionally, an alert would pop up to tell you something, text in a box that delivered news would scroll, or a cookie would store your name and display it back to you when you returned months later. There were surely no jobs where JavaScript was the primary language, save those lucky few who had the job of actually creating JavaScript. In short, it was a gimmick for websites to fancy up their DOM.

Today, JavaScript can be found virtually everywhere. From AJAX, to Bootstrap, to React, AngularJS, the ubiquitous jQuery, and even Node.js ▶

on the server side, JavaScript has become one of the most important and popular web languages.

Frameworks

One of the biggest ways JavaScript has changed since its inception is in its use. Gone are the days of awkward document.GetElementById calls and cumbersome XmlHttpRequest objects. Instead, helpful libraries have abstracted much of the root functionality, making JavaScript more accessible to developers. This is a big part of why you see JavaScript everywhere.

jQuery

jQuery was created in 2006 by John Resig. It provides a rich set of tools that abstract and simplify cryptic and rigid JavaScript commands and methods. The easiest way to demonstrate this is by example of an AJAX request made with vanilla JavaScript: (Code 1)

And here is the same AJAX request made with jQuery (Code 2).

```
function loadXMLDoc() {
    var xmlhttp;

    if (window.XMLHttpRequest) {
        // code for IE7+, Firefox, Chrome, Opera, Safari
        xmlhttp = new XMLHttpRequest();
    } else {
        // code for IE6, IE5
        xmlhttp = new ActiveXObject(“Microsoft.XMLHTTP”);
    }

    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == 4 ) {
           if(xmlhttp.status == 200){
               alert(“success”);
           }
           else if(xmlhttp.status == 400) {
              alert(“error 400”)
           }
           else {
               alert(“something broke”)
           }
        }
    }

    xmlhttp.open(“GET”, “test.html”, true);
    xmlhttp.send();
}
```

Code 1 / Source: Stack Overflow

```
$.ajax({
    url: “test.html”,
    statusCode: {
         200: function() {
                alert(“success”);
         },
         400: function() {
                alert(“error 400”);
         }
    },
    error: function() {
         alert(“something broke”);
    }
});
```

Code 2

jQuery made difficult JavaScript functions easy to use and DOM manipulation a snap. As a result, it was one of the first widely used frameworks for JavaScript, and the idea of abstraction that came with it was the basis upon which other frameworks were modeled.

AngularJS

AngularJS, typically called simply "Angular", hit the scene in 2009. It was created by Google to make it much easier to build single-page applications (SPAs). Like jQuery, it aims to abstract the difficult work into highly reusable methods. It provides a model-view-controller (MVC) architecture for JavaScript.

React

React is new to the game. It was created by Facebook and released for the first time in 2013. Facebook considers React to be a new take on the SPA problems that Angular works to solve. You would be correct to consider Angular and React as competing frameworks but what really separates React from Angular is that React is a more efficient, higher-performance, quantifiably faster library. The below chart shows the time taken by React, Angular, Knockout (a library not discussed in this article), and vanilla JavaScript to render a list of 1,000 items to the DOM (Image 1)

If performance is important to your application, React is the way to go.

The JavaScript development environment

An important part of efficient development is the use of an integrated development environment (IDE). An IDE is an application that offers a suite of tools to a developer. The most important of these tools is typically a rich-text editor, which often offers syntax highlighting, autocomplete, and keyboard shortcuts that speed up annoying manual processes.

Sublime Text

Sublime Text is not actually an IDE. It is a lightweight, super-fast, programming text editor that offers syntax highlighting and intuitive keyboard shortcuts. It's cross-platform, which is excellent for developers who want to use a Mac in a Windows shop or vice versa. Virtually everything about Sublime Text can be customized. It also offers multiple plugins that enable IDE-like capabilities such as Git integration and linting. This is a terrific choice for enthusiasts ▶

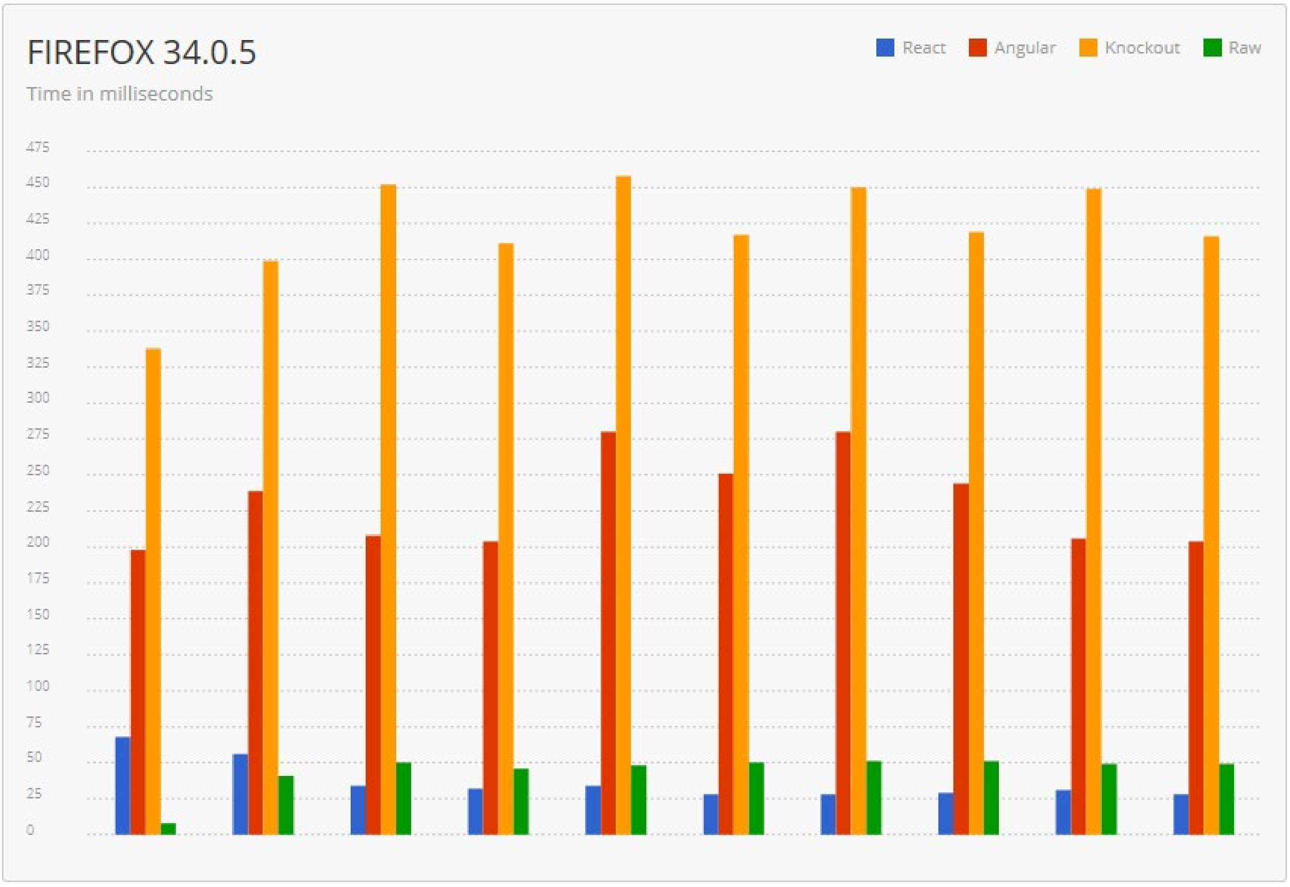
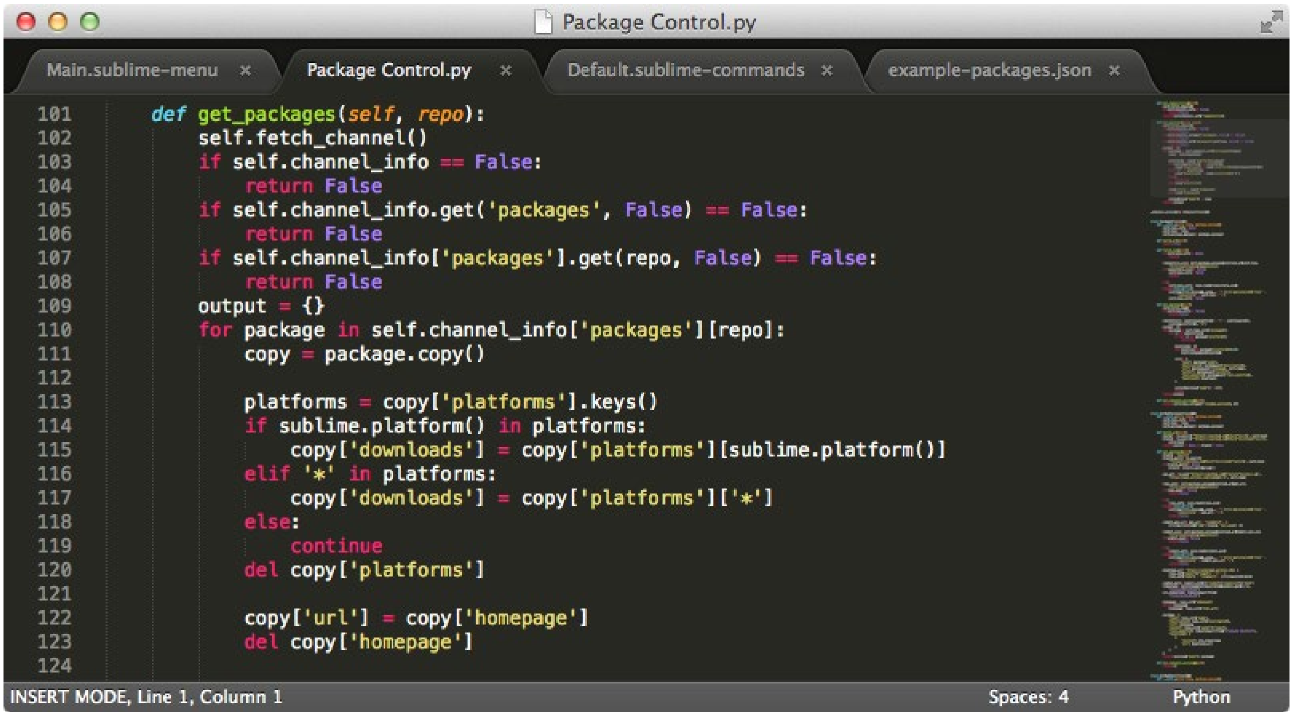


Image 1 / Source: The Dapper Developer



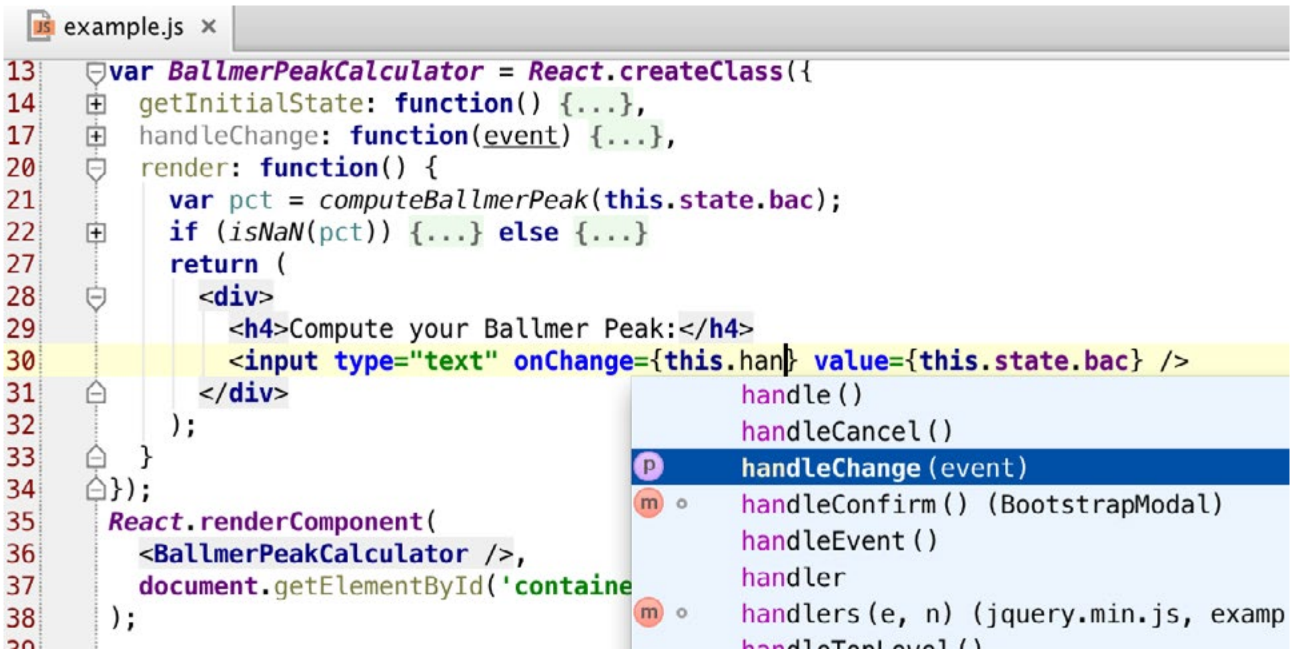
```python
    def get_packages(self, repo):
        self.fetch_channel()
        if self.channel_info == False:
            return False
        if self.channel_info.get('packages', False) == False:
            return False
        if self.channel_info['packages'].get(repo, False) == False:
            return False
        output = {}
        for package in self.channel_info['packages'][repo]:
            copy = package.copy()

            platforms = copy['platforms'].keys()
            if sublime.platform() in platforms:
                copy['downloads'] = copy['platforms'][sublime.platform()]
            elif '*' in platforms:
                copy['downloads'] = copy['platforms']['*']
            else:
                continue
            del copy['platforms']

            copy['url'] = copy['homepage']
            del copy['homepage']
```

```
INSERT MODE, Line 1, Column 1                    Spaces: 4        Python
```

Image 2 / source: Sublime Text



```javascript
var BallmerPeakCalculator = React.createClass({
  getInitialState: function() {...},
  handleChange: function(event) {...},
  render: function() {
    var pct = computeBallmerPeak(this.state.bac);
    if (isNaN(pct)) {...} else {...}
    return (
      <div>
        <h4>Compute your Ballmer Peak:</h4>
        <input type="text" onChange={this.han} value={this.state.bac} />
      </div>
    );
  }
});
React.renderComponent(
  <BallmerPeakCalculator />,
  document.getElementById('containe
);
```

```
handle()
handleCancel()
handleChange(event)
handleConfirm() (BootstrapModal)
handleEvent()
handler
handlers(e, n) (jquery.min.js, examp
```

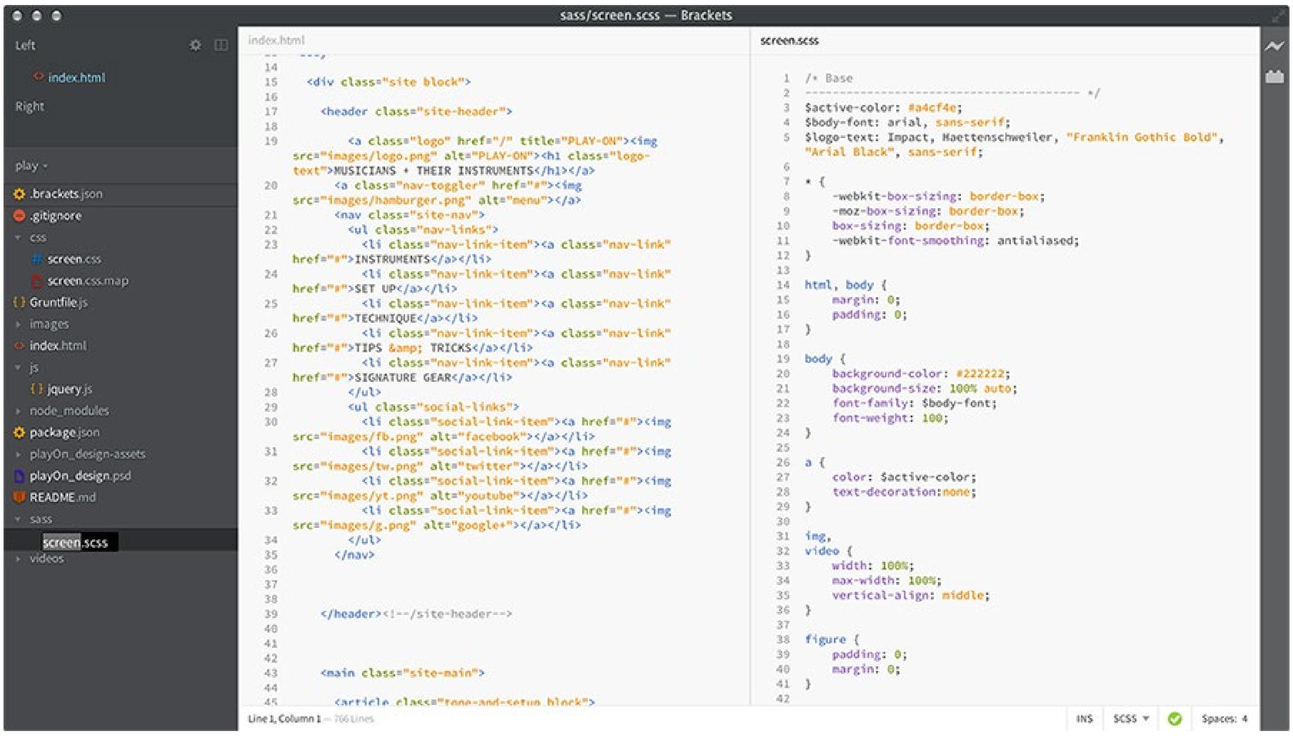
Image 3 / Source: JetBrains

and new JavaScript developers alike. A Sublime Text license costs US$70 at the time of writing. (Image 2)

WebStorm

WebStorm was created by the JetBrains team as a smart IDE focused on HTML, CSS, and JavaScript. It costs US$49, plus another US$29 for upgrades each year beyond the first (for individuals). WebStorm is widely considered the de facto standard for seasoned JavaScript professionals, and for good reason. The built-in code-completion and inspection tools are second to none. WebStorm also offers a rich JavaScript debugger and integrated unit testing with popular frameworks such as the Karma test runner, JSDriver, and even Mocha for Node.js.

One of the nicest features of WebStorm is the LiveEdit functionality. By installing a plugin into both Chrome and WebStorm, a developer can make source-code changes that are instantly reflected in the browser. Developers can also configure LiveEdit to highlight the changes that are made in the browser window, making both debugging and coding highly productive.



```html
  <div class="site block">

    <header class="site-header">

        <a class="logo" href="/" title="PLAY-ON"><img
src="images/logo.png" alt="PLAY-ON"><h1 class="logo-
text">MUSICIANS + THEIR INSTRUMENTS</h1></a>
      <a class="nav-toggler" href="#"><img
src="images/hamburger.png" alt="menu"></a>
      <nav class="site-nav">
        <ul class="nav-links">
          <li class="nav-link-item"><a class="nav-link"
href="#">INSTRUMENTS</a></li>
          <li class="nav-link-item"><a class="nav-link"
href="#">SET UP</a></li>
          <li class="nav-link-item"><a class="nav-link"
href="#">TECHNIQUE</a></li>
          <li class="nav-link-item"><a class="nav-link"
href="#">TIPS &amp; TRICKS</a></li>
          <li class="nav-link-item"><a class="nav-link"
href="#">SIGNATURE GEAR</a></li>
        </ul>
        <ul class="social-links">
          <li class="social-link-item"><a href="#"><img
src="images/fb.png" alt="facebook"></a></li>
          <li class="social-link-item"><a href="#"><img
src="images/tw.png" alt="twitter"></a></li>
          <li class="social-link-item"><a href="#"><img
src="images/yt.png" alt="youtube"></a></li>
          <li class="social-link-item"><a href="#"><img
src="images/g.png" alt="google+"></a></li>
        </ul>
      </nav>



    </header><!--/site-header-->



    <main class="site-main">

      <article class="tone-and-setup block">
```

```scss
/* Base
---------------------------------------- */
$active-color: #a4cf4e;
$body-font: arial, sans-serif;
$logo-text: Impact, Haettenschweiler, "Franklin Gothic Bold",
"Arial Black", sans-serif;

* {
    -webkit-box-sizing: border-box;
    -moz-box-sizing: border-box;
    box-sizing: border-box;
    -webkit-font-smoothing: antialiased;
}

html, body {
    margin: 0;
    padding: 0;
}

body {
    background-color: #222222;
    background-size: 100% auto;
    font-family: $body-font;
    font-weight: 100;
}

a {
    color: $active-color;
    text-decoration:none;
}

img,
video {
    width: 100%;
    max-width: 100%;
    vertical-align: middle;
}

figure {
    padding: 0;
    margin: 0;
}
```

Image 4 / Source: Brackets

Overall, WebStorm is the IDE to pick if JavaScript is your full-time job. (Image 3)

Brackets

Brackets is an open-source, free IDE built with a focus on visual tools. Brackets offers a live editing feature similar to WebStorm that lets you instantly see the results of code changes in a browser window. It also supports side-by-side editing, which allows you to work on the code and see the results simultaneously without the need for tabbing between applications or pop-up windows. One of the most interesting features of Brackets is called Extract. It analyzes Photoshop (PSD) files in order to retrieve font, color, and measurement information. This feature makes Brackets an excellent choice for JavaScript developers who also do design work. (Image 4)

Atom

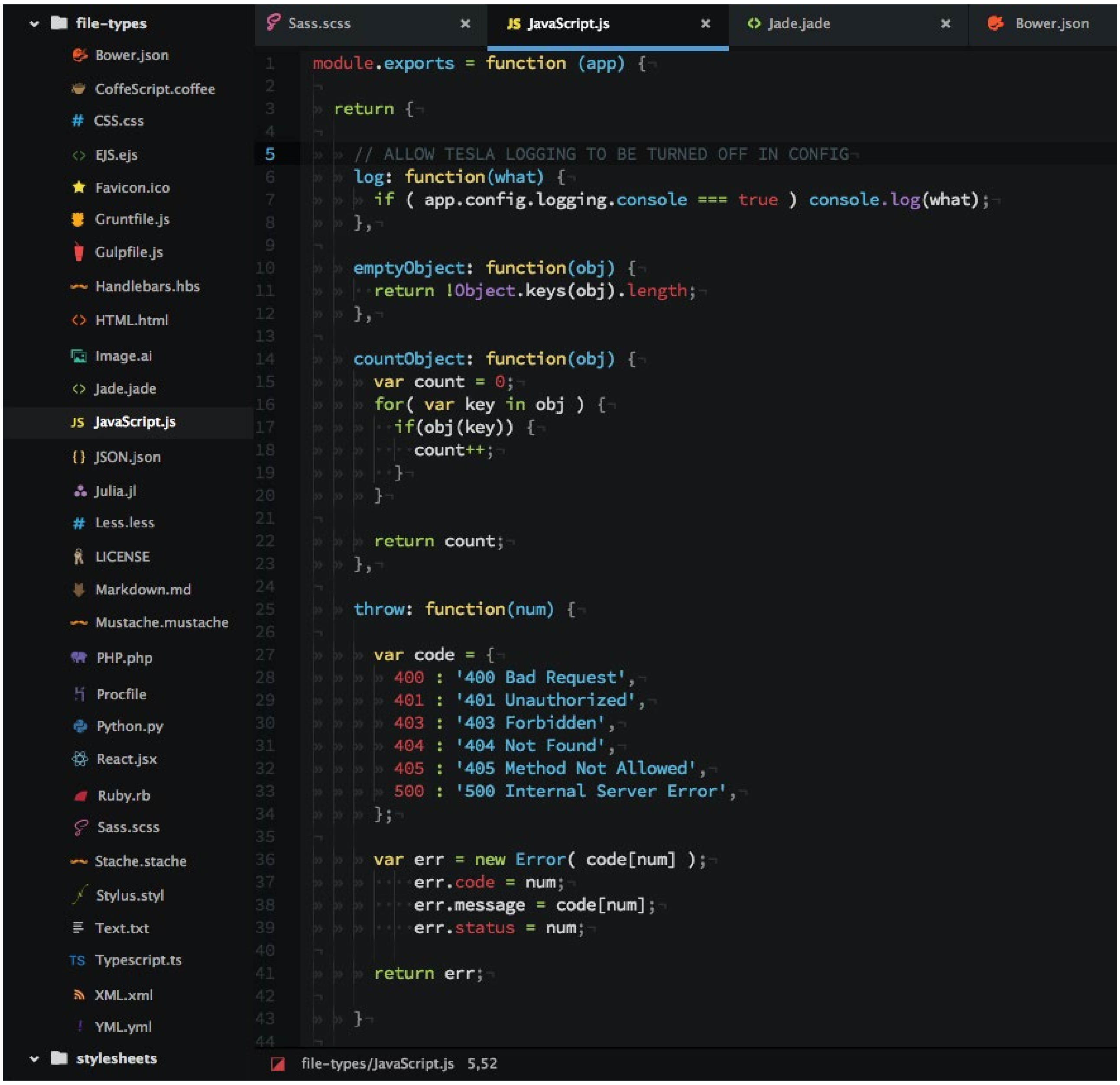
Atom is an open-source, free rich text editor created by GitHub. It is very approachable and easy to use; you can install and run it without ever touching a configuration file and it just works. The most interesting part of Atom is its ability to customize all of its aspects (GitHub calls it "hackable"). Atom is built upon a web core, enabling you to customize its look and feel with standard HTML, CSS, and JavaScript. Want a different background and text color in Atom? Just change the CSS values. Alternatively, you can download and apply one of many themes created for Atom. This allows Atom the flexibility to become anything that you'd like it to be. Atom is an excellent tool for new JavaScript developers and enthusiast hackers alike. (Image 5)

Build tools and automation

The modern JavaScript project tends to be fairly complex, with many moving parts. This is not due to inefficiencies in the language or tools; it is a direct result of the rich, vibrant, and complex web applications that are being built today. When working on a large project, there will be many repetitive processes that you must do whenever you want to check in your code or build out to production. These could be things like bundling, minification, compilation of LESS or SASS CSS files, and even running tests. Doing this work manually is frustrating and inefficient. It's better to automate the work via a build tool that supports tasks.

Bundling and minification

Most of the JavaScript and CSS that you write will be shared across a few web pages. As a result, you will likely place them in .js and .css files, and then reference those files on your web page. This will cause the visitor's browser to make an HTTP request ►



```
module.exports = function (app) {

    return {

        // ALLOW TESLA LOGGING TO BE TURNED OFF IN CONFIG
        log: function(what) {
            if ( app.config.logging.console === true ) console.log(what);
        },

        emptyObject: function(obj) {
            return !Object.keys(obj).length;
        },

        countObject: function(obj) {
            var count = 0;
            for( var key in obj ) {
                if(obj(key)) {
                    count++;
                }
            }

            return count;
        },

        throw: function(num) {

            var code = {
                400 : '400 Bad Request',
                401 : '401 Unauthorized',
                403 : '403 Forbidden',
                404 : '404 Not Found',
                405 : '405 Method Not Allowed',
                500 : '500 Internal Server Error',
            };

            var err = new Error( code[num] );
                err.code = num;
                err.message = code[num];
                err.status = num;

            return err;

        }
```

Image 5 / Source: Atom

to retrieve each of these files (or at least verify that they haven't changed) in order to fully render your web app.

HTTP requests are expensive. On top of the payload size, you also pay the costs of network latency, headers, and cookies. Bundling and minification tools are designed to reduce or eliminate these costs entirely.

Bundling

One of the simplest things that a developer can do to improve the performance of their web code is to bundle it. Bundling is the process of stitching multiple JavaScript or CSS files into a single JavaScript or CSS file. This is just like stitching the individual pieces of a panoramic photo together at the seams in order to create a single, contiguous photo. By bundling our JavaScript and CSS files, we eliminate much of the HTTP request overhead.

Minification

Another way that a JavaScript developer can improve performance is by minifying their newly bundled code. Minification reduces JavaScript and CSS to their smallest possible forms while maintaining full functionality. For JavaScript, this means renaming variables to nonsensical single-character tokens, and removing all white space and formatting. For CSS, which relies on the names of variables, this typically means only removing formatting and white

```
$.ajax({
    url: “test.html”,
    statusCode: {
        200: function() {
            alert(“success”);
        },
        400: function() {
            alert(“error 400”);
        }
    },
    error: function() {
        alert(“something broke”);
    }
});
Here’s the same code minified:
$.ajax({url:”test.html”,statusCode:{200:function(){alert(“success”);},
400:function(){alert(“error 400”);}},error:function(){alert(“something broke”);}});
```

Code 3

space. Minification drastically improves network performance because it reduces the number of bytes sent in each HTTP response.

Look at our un-minified segment of AJAX JavaScript code from earlier: (Code 3)

Note that I broke the minified output into two lines to display it in this article. The actual output of minification is typically a single line.

Usually, the bundle and minify steps are only done in production. This is so that you can debug your code locally or in a development environment as is, complete with formatting and line numbers. Debugging the minified code above would be difficult: all of the code would be on line 1. Minification makes the code completely unreadable, which would be useless and frustrating for you to attempt to debug.

Source map files

Sometimes a bug in your code occurs that is only reproducible in production. This poses a problem when you need to debug the issue but all of your code is minified. Fortunately, JavaScript supports source-code map files that map the minified code to the original code. The map file for your code is generated during minification via build tools such as those described below. Your favorite JavaScript debugger then uses the map file to provide you with legible code to debug. You should deploy your production code with map files whenever possible, so that you can debug the code if anything goes wrong.

Linting

A linting tool analyzes your code for common mistakes and deviations from defined formatting rules. The kinds of errors reported are things like using tabs instead of spaces, missing semicolons at the ends of lines, or curly braces without an if, for, or while declaration. Most IDEs come with linting tools; others offer the ability to install a linting plugin.

The most popular JavaScript linters are JSHint and JSLint. JSHint is developed by the community and is a fork of JSLint, the original linting framework built by Doug Crockford. These two linters vary a little in the format standards they enforce. My advice is to try both and use whichever one best fits your style of coding.

Automating things: Grunt

Despite the name, Grunt is far from primitive. It is a robust command-line build tool that executes user-defined tasks. By specifying a simple configuration file, you can configure Grunt to compile LESS or SASS, build and minify all of the JavaScript and CSS files in specific folders, or even run a linting tool or test framework. You can also configure Grunt to execute as part of a Git hook that minifies and bundles your code whenever you check something into the source-control repository.

Grunt supports named targets so that you can specify different commands for different environments; you could define “dev” and “prod” as targets, for example. This is useful for scenarios such as bundling and minifying your code in production but leaving it alone in the development environment so that it is easy to debug.

A useful feature of Grunt is “grunt watch”, which monitors a directory or set of files for changes. This can be integrated directly with IDEs such as WebStorm and Sublime Text. With grunt watch, you can trigger events based on file changes. A ▶

practical application of this approach is LESS or SASS compilation. By configuring grunt watch to monitor your LESS or SASS files, you can compile them immediately whenever a file is changed, making the output of the compilation immediately available to your development environment. You could also use grunt watch to automatically run a linting tool against any file that you edit. Real-time execution of tasks via grunt watch is a terrific way to boost your productivity.

Automating things: Gulp

Grunt and Gulp are direct competitors that strive to solve the same build-automation problems. The major difference between Grunt and Gulp is that Grunt focuses on configuration while Gulp focuses on code. While you'd configure build tasks via declarative JSON in your Grunt file, you'd write JavaScript functions in your Gulp file to accomplish the same.

This is a Grunt file configured to compile SASS files to CSS whenever a file changes:

```
grunt.initConfig({
  sass: {
    dist: {
      files: [{
        cwd: “app/styles”,
        src: “**/*.scss”,
        dest: “../.tmp/styles”,
        expand: true,
        ext: “.css”
      }]
    }
  },
  autoprefixer: {
    options: [“last 1 version”],
    dist: {
      files: [{
        expand: true,
        cwd: “.tmp/styles”,
        src: “{,*/}*.css”,
        dest: “dist/styles”
      }]
    }
  },
  watch: {
    styles: {
      files: [“app/styles/{,*/}*.
scss”],
      tasks: [“sass:dist”,
“autoprefixer:dist”]
    }
  }
});
grunt.registerTask(“default”,
  [“styles”, “watch”]);
```

Source: Jack Hsu

This is a Gulp file configured to compile SASS files to CSS whenever a file changes:

```
gulp.task(“sass”, function () {
  gulp.src(“app/styles/**/*.scss”)
    .pipe(sass())
    .pipe(autoprefixer(“last 1
  version”))
    .pipe(gulp.dest(“dist/styles”));
});
gulp.task(“default”, function() {
  gulp.run(“sass”);
  gulp.watch(“app/styles/**/*.scss”,
  function() {
    gulp.run(“sass”);
  });
});
```

Source: Jack Hsu

Use whichever one you prefer. Both of these tools are typically installed via npma, a Node.js package manager.

In summary

JavaScript has evolved significantly since its birth amidst the early days of the Web. Today, it is a prominent and important feature of interactive web applications.

The developer has also evolved significantly since 1995. Today's modern JavaScript developer employs rich and robust frameworks, tools, and IDEs to work efficiently and productively.

Building your first modern JavaScript application is easier than you may think! Just choose an IDE (I recommend Atom for beginners), and then install npm and Grunt. If you get stuck along the way, Stack Overflow is an excellent resource. With just a little time spent learning the basics, you'll be well on your way to releasing your first modern JavaScript app. ■

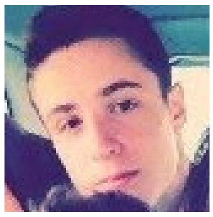
READ ONLINE ON InfoQ

Virtual Panel: Real-World JavaScript MVC Frameworks

by Dio Synodinos

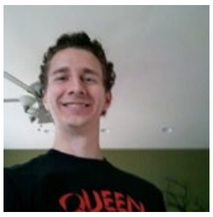
THE PANELISTS

Igor Minar is a software engineer at Google. He is the lead of AngularJS, a practitioner of test-driven development, an open-source enthusiast, and a hacker.



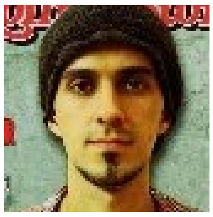
Matteo Pagliazzi is a passionate software developer and an open-source contributor.

Julien Knebel is a self-taught interface designer and a front-end developer living in Paris where he is freelancing for some of the biggest French companies.



Brad Dunbar is a JavaScript programmer and a frequent contributor to the Backbone.js and Underscore projects.

John Munsch is a professional software developer with over 27 years of experience. These days, he's leading a team building modern web-app front ends with AngularJS after a couple of years spent doing the same kind of work with Backbone.js, Underscore, and Handlebars.

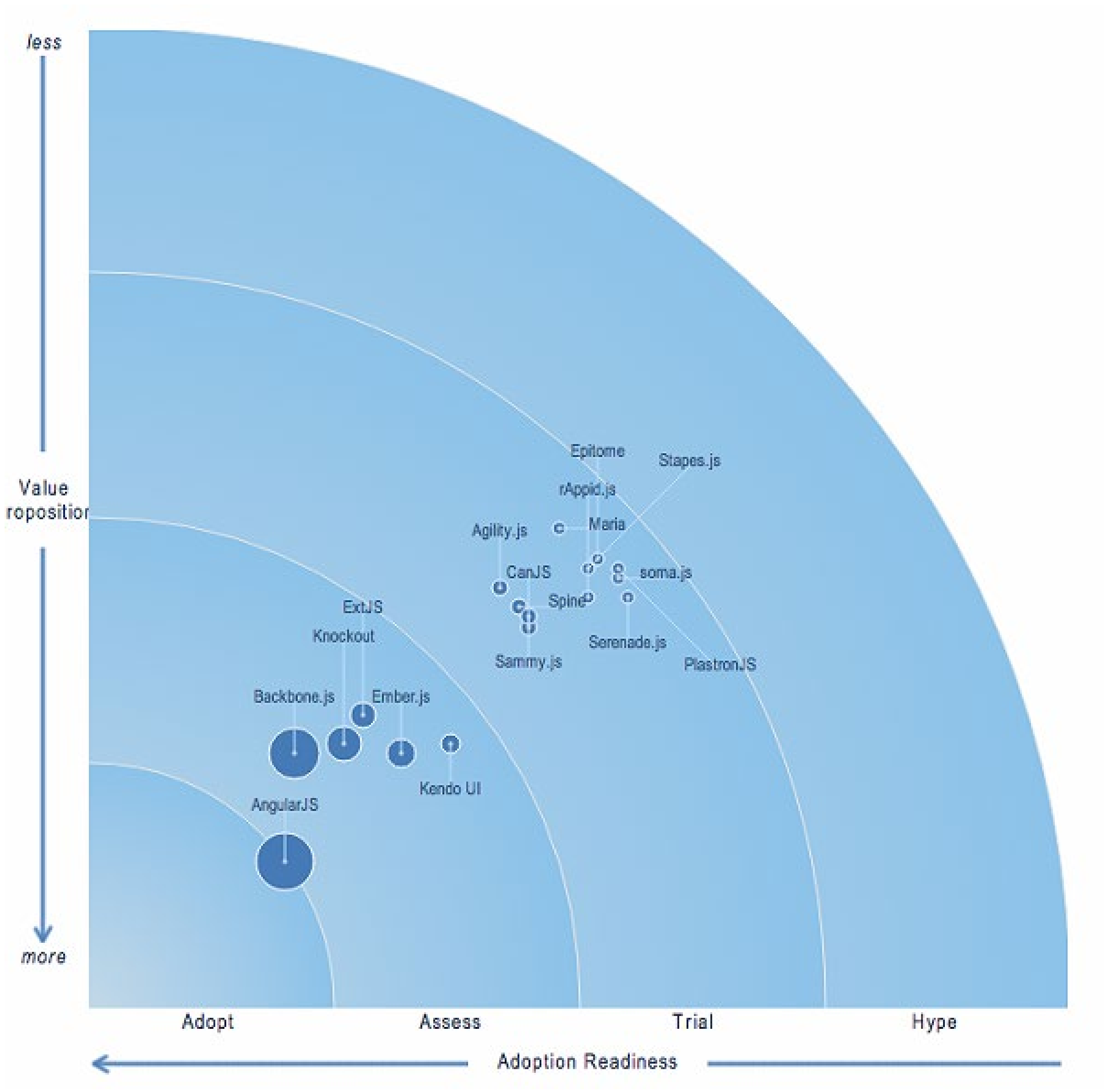


Julio Cesar Ody is a software developer, designer, presenter, and wannabe writer who lives in Sydney. He works a lot with mobile web development,and has a built a handful of tools which he's very proud of.

Thomas Davis is the founder of Backbone Tutorials, co-founder of CDNJS, and a developer for Taskforce. He's also a daily contributor to many other open-source projects, which can be found at github.com/thomasdavis.

As the browser end up executing more and more logic, JavaScript front-end codebases grow larger and more difficult to maintain. As a way to solve this issue, developers have been turning to MVC frameworks, which promise increased productivity and maintainable code.

Back in 2013, InfoQ had asked the community to rank JavaScript MVC frameworks according to the features they had and how mature they were. You can see the result in the following figure.



InfoQ has asked a handful of expert practitioners about how they use these frameworks and the best practices they follow when developing JavaScript applications.

What is your favorite JavaScript MVC framework, how long have you been using it, and what is the fundamental problem that it helps you with?

Matteo Pagliazzi: AngularJS is actually the framework of the moment, and it's not only a matter of GitHub stars: it has the biggest number of libraries, modules, etc. and it's being talked about (and used) by a lot of developers. It's obvious that we (devs) like it. This is, I think, because it's very easy to get started with. It takes five minutes to have something even not trivial working but at the same time it's very powerful. So the learning curve gets steep as you move from the basic things to the core of the framework, when you find out it's very complex.

Julien Knebel: I've been developing with Ember.js for more than two years now on five production web apps. I'd say that it gave me flexibility, speed, and confidence so I could focus more on shaping the whole user experience instead of investing huge amount of time in complex features, like a robust asynchronous router for instance.

Brad Dunbar: My favorite is Backbone.js. It provides simplicity and flexibility that others don't. Also, perhaps more importantly, its code is generally simple and readable.

John Munsch: My favorite framework is AngularJS, I've been using it just over a year and the honeymoon is not over yet.

The fundamental problem I look for any front-end framework to solve is to give me something with sufficient bones, muscle, and connective tissue to help me build a complex UI on the front-end not just comparable to but better than what we could have built with a request/response back-end framework like JSP, PHP, Rails, etc. in the past.

Julio Cesar Ody: I mostly use Backbone.js, and increasingly React.js has replaced Backbone.js views for me.

All web applications I build run fully in the browser, and communicate with a server using an API and/or WebSockets. This means I end up writing a lot of logic in JavaScript, so Backbone.js helps me keep things orderly and maintainable.

Thomas Davis: My first experience with formalized JavaScript MVCs was late 2010 and at the time I was deciding between using Backbone.js and SproutCore. I posted a comparison of the two, which reached the front page of Hacker News and received over 100,000 views. Based on the feedback at the time, I decided to use Backbone.js because it was minimal and didn't stop me from implementing complex functionality without being confined by the conventions of a larger framework. As I learned Backbone.js, I wrote tutorials to give myself a more concrete understanding of the framework. These posts have received millions of views and still do. The current archives are at backbonetutorials.com. When building a product these days, the user experience and design is as crucial as the necessity of keeping the site up. Single-page applications allow you to make user experiences that traditional pages cannot match due to page reloading. For example, imagine using Facebook if every like and every comment made the page refresh.

How does the framework you use compare with the many alternatives?

Julien Knebel: Ember.js performs really well! I never imagined it would let me code entire apps almost all by myself. AngularJS is also really good though; I really like how fast you can get up and running with it. However, I believe it tends to become more and more tedious as your codebase grows. On the other hand, I had more difficulties at the beginning with Ember.js but they started to melt like snow in the sun after a couple of "aha!" moments while learning it. Another important point (at least for me) is "componentization". I disliked how Angular Directives deals with this and Ember Components really feel like the right way of doing it.

Brad Dunbar: I find that it's very rare that a JavaScript library can precisely fill all the needs of my application. This means that I often end up writing around them. Backbone's functionality is small and provides utilities that I can compose instead of larger pieces that aren't exactly what I need.

John Munsch: That's a tough one; my experience only stretches to Backbone.js (just under two years' experience) and AngularJS (just over a year). But between those two, I don't think it's a tough choice at all.

Backbone.js was okay, but we had a large project with as many as half a dozen people working on it at its peak and the problem we got into was with all of the large gaps in Backbone.js that you typically fill with other software (two-way binding, validation, templating, AMD, etc.). If you don't figure out your solutions for all of those gaps before you start, every developer in the group ends up filling it his or her own way. If you're under a time crunch and writing really fast, it can be really difficult to catch all of the divergence and enforce much consistency. We ended up with a working project at the end and it was way faster than the old solution but it was a chimera; every part of it appeared to be a different animal. With AngularJS, a lot more of those parts are ►

part of the framework itself and there's less room for that kind of problem.

Julio Cesar Ody: I'd say Backbone.js is the most minimal, but that's not a particularly useful point to focus on. Ultimately, what one resonates with the most and, in a way, the one that you're happier working with is also the one you're going to be most productive with.

So between Backbone.js, AngularJS, and Ember.js, you can't go wrong. I'm leaving React.js out since it's really just a UI components library, so it shouldn't be confused with an applications framework.

Thomas Davis: A lot has changed since 2010 and even though I am always trying to break out of my old habits, I still use Backbone.js at the moment. It has remained minimal and hasn't changed any of its core beliefs or methods. Though in my opinion, it definitely is on the verge of being superseded by the competition. Efforts made by libraries such as AngularJS to give developers more intuitive control over the DOM make Backbone.js seem archaic when it comes to rendering. With the new modular JavaScript movement, it is also possible to call Backbone.js monolithic in size. There isn't any reason why Models, Collections, Routers, and Views should be packaged together into one library; alternatives such as Components.js are writing those parts as individual modules. This hasn't actually been done fully at the moment and I'm considering it for one of my next projects.

Building faster, more robust front ends is nice but what about performance, especially on mobile?

Igor Minar: I think mobile is still underserved for web development. A lot more focus needs to be placed here and AngularJS is definitely refocusing on mobile in its 2.0 version.

Julien Knebel: Ember.js is heavier than others – it's a fact, even if you minify/gzip it. This is important to consider especially if you plan to build mobile web apps. Then, dealing with smooth 60-FPS transitions is tough:; either you implement them with Ember.js or another one. I'd say that you need to know how to deal with common rendering performance issues, otherwise you'll get in trouble no matter what framework you choose.

Brad Dunbar: Backbone.js is rather performant and small compared to others. I haven't had any issues using it for mobile, though I usually don't write for devices more than one generation old.

John Munsch: We haven't really focused on mobile yet for my current site, but we definitely did at my previous employer. Because they needed to run an additional web interface on barcode guns in warehouses and hangars, we crafted a secondary UI which was a handful of pages built just for the guns. They used the same technology as the desktop browser UI and the pages called the same back-end API but the pages were smaller and simpler for the small touch screens on the guns.

In that environment, it seemed a better solution than trying to make a responsive version of existing pages which would need to drop 80% of their functionality on mobile. (We were very lucky that good-quality barcode guns with HTML5 browsers became available before the project was due to ship. Early work was done with guns that only had IE 5.5 available. *shudder*)

If I were to return to this project today, the first thing I would probably do is add Grunt for concatenation, minification, and revisioning of the JavaScript to speed up the mobile and desktop further and turn on expires headers so browser caching could be turned up to a month or more.

Julio Cesar Ody: JavaScript is fast and browsers are pretty damned efficient these days and increasingly becoming more so. It does take some serious messing up for things to end up really slow.

That said, performance is everything when it comes to good UX. But the JavaScript side of things is just one part of that. How you use CSS (and transitions, animations), how good and thin the markup is – all that also has a big impact on performance. You need to be mindful about the whole.

If any of you did any microcontroller programming in C, perhaps you know what it means to write programs that run on constrained devices. Although phones are rather powerful computers, they're still significantly slower than the average laptop. You can't afford to make as many mistakes, and you need to take the rule I mentioned before and be twice as careful.

Measuring problems helps a lot. Chrome DevTools comes to mind.

Thomas Davis: There is no silver bullet for choosing a front-end-development approach between mobile and desktop at the moment. Sometimes native apps are better, sometimes JavaScript apps are, and sometimes server-generated pages. Though for performance questions related to the DOM in general, I'd probably make a safe bet on React paving the way. React implements a virtual DOM that diffs the browser's DOM and only makes changes when necessary for high-performance rendering. Because it's a virtual DOM, you can also render DOMs on the server and access them programmatically.

What is your typical workflow using your framework of choice? What tools do you use to develop and debug?

Igor Minar: WebStorm, Karma, Chrome.

Julien Knebel: I use Grunt as my build tool to preprocess, precompile, post-process, concatenate, minify, etc. Then TextMate 2 has been my IDE of choice for years. I also spend a huge amount of time in Chrome DevTools debugging with breakpoints and debuggers. And of course I couldn't live without Git.

Brad Dunbar: I tend to use a somewhat minimalist setup including a CLI and vim/node/npm. Lately, I've enjoyed using Browserify for bundling. As for workflow, I do the typical code, refresh, debug cycle. I've never been a huge proponent of test-first or anything of that nature.

John Munsch: Most of the developers on the front-end team use WebStorm or Sublime on Macs. The project as a whole is built and run via Maven because the back end is Java, and we recently started running Grunt from within our Maven builds to get optimizations on front-end code. Jenkins is used to build, run unit tests, and deploy to various testing environments and production.

Local developers run Karma in the background to run the AngularJS unit tests we have (sadly only 30% code coverage today).

Julio Cesar Ody: I do as much of the design as I can (when I'm the one doing it) up front. Looking at how the page will look like helps me do a mental breakdown of the components I'll need to write.

I've used [Hopla](), which I wrote myself, a lot recently as a build tool. It uses Sprockets for transpiling CoffeeScript and SASS, and compiles the application to static HTML/CSS/JS. This comes in handy for deployments, and even for building PhoneGap apps.

Then I start implementing a design usually by writing a static HTML page with styles using SCSS, going through each part of it later and replacing it with JavaScript components.

Thomas Davis: Honestly, I'm pretty indifferent to how people like to develop and debug, and my methodologies change on a per-project basis. The only advice I dogmatically give is that developers use Asynchronous Module Definition (AMD). RequireJS gives a rock-solid implementation of AMD and in my experience makes debugging the easiest of all languages and environments. Not only does it help you structure your code intelligently but it also makes the barrier to entry of your codebase very low because everything has to be a dependency referenced by a file path.

As applications grow bigger, it can be a challenge to keep a robust architecture and maintain a large codebase. How does your favorite JavaScript framework scale? How does it scale as development teams become bigger and different people need to work on different parts of the functionality?

Igor Minar: Code reuse, reduction of boilerplate and style guide, and conventions are important for reducing complexity of large codebases. But in our real-world experience, we've seen that high-quality test suites have even higher impact because they allow for major refactoring while keeping the risk low. And refactoring is a key to keeping the codebase clean.

Matteo Pagliazzi: Besides the complexity (like the presence of Services, Factories, and Providers, which is very confusing), what I personally don't like is the dirty-checking mechanism AngularJS uses to see which properties have changed and update the view accordingly (but with simplicity a goal of [AngularJS 2.0](), this is going away as Object.observe is implemented natively, allowing for notifications to ▶

READ IT NOW!
Best news from the industry curated by professionals like you.
READ LATER
now!
InfoQ queue

be issued upon change instead of having to check for every property's changes).

To wrap up, AngularJS is really great, backed by an awesome community and an enormous number of external libraries that usually satisfy every need – and if you don't find what you need, you can always write your own directive. But it is also full of complex concepts that take time to understand.

Julien Knebel: Ember.js deals really well with growing codebases because of its enforced good practices and its strong conventions, which let you do a lot with single lines of code (a.k.a. convention over configuration). It helps each member of the team debug others' code faster. I saw some developers getting angry because they felt constrained to code how Ember.js wanted them to code, but I guess the real frustration came from the fact that you first have to understand/learn Ember.js before trying to do fancy stuff with it. And I do believe that this extra education is time really worth it in the long term. Plus, I tend to trust Yehuda Katz and Tom Dale's work a lot (but this is not really objective, I guess).

Brad Dunbar: I think large JavaScript codebases are challenging regardless of the framework used. It comes down to the team maintaining a set of guidelines and sticking to them. Use of modules of some kind (I mentioned I like npm and Browserify) helps tremendously though.

John Munsch: So far, our experiences with scaling are good. Front-end JS scales well because there are lots of static files amenable to optimization, caching, and even CDNs. Pretty much the only bad experiences have been with trying to render thousands of items worth of data on pages. We've tried to encourage the use of pagination and other workarounds but we can't always get agreement to use them and large chunks of data generating thousands of DOM elements is a recipe for problems on old browsers.

Other problems have actually completely gone away. For example, AngularJS's default behavior of discarding views and controllers when switching between views keeps memory usage of another view from impacting the present view. That's a simple solution that helps developers avoid a lot of problems as they add more and more functionality to a single-page app.

Julio Cesar Ody: I don't think any of the popular frameworks has a defined scaling path. It's up to the developer to think of something sane and run with it.

I've always liked the concept of components/modules. It lets me think of writing programs much like building a watch. Each part (or component) has its purpose, and needs to work as independently as possible, exposing a small surface area that other components can interact with.

Thomas Davis: As long as you use a modular JavaScript framework such as RequireJS, scaling and maintaining your codebase is a walk in the park and will only depend on your coding practices. Though, Backbone.js does need to be split into its own modules at this point so instead of requiring Backbone as a dependency, you would only require Backbone.Model, for example.

What common pitfalls would a team that is only now considering adopting a JavaScript framework have to worry about?

Matteo Pagliazzi: Working on a big open-source project that uses AngularJS as the client-side framework, I found that developers without much experience easily incur problems with inheritance or start polluting the $scope and $rootScope with a lot of objects that at a certain point will start to slow the application and grow the RAM used by the browser (we easily use 300+ MB and it can easily get to 1 GB with an even small memory leak). The fact that it's made data binding simpler is of course fantastic but you have to understand that it will not always work as you want just because it's magic.

Julien Knebel: All those MV* frameworks are heavy JavaScript calculations on the front end, therefore if your app needs to print a large amount of data to the user, you'll quickly hit your performance budget (talking about the 60-FPS budget). You'll have to deal with pagination, intelligent scrolling, wisely dosed CSS decorations, subtle sequencing, choose where to display spinners, choose when to let network calls retrieve data, etc. Eventually, all that stuff will matter a lot.

Brad Dunbar: I think the most common issues come before you've even dug into the framework. Your client needs tests, modules, a package manager, and continuous integration just like a server. If you can stay true to those, you'll probably be all right.

John Munsch: I've mentioned the problems we had with Backbone.js, but there are some common things that can always come up. For example, SEO is a factor for pages generated entirely via JavaScript on the front end. I've been fortunate that most of my work has been SaaS so SEO hasn't been anything we've cared much about, but if you're building something for the wider Web, you need to look at solutions like Prerender.io, BromBone, etc. to see how you're going to deal with that. Google Analytics can present similar problems. None of these are unsolvable, but it's good to go in knowing that there are some issues and you may need to pick a solution.

For us, without a doubt, IE8 has been one of our biggest problems. It's the last browser with a significant chunk of market share out there that does not have a built-in JavaScript just-in-time compiler. As a result, JavaScript-heavy front ends can be slow at times, can trigger "unresponsive script" errors when displaying lots of data, etc. The sooner the Lovecraftian horror that is IE8 goes away, the better for all of us.

Julio Cesar Ody: The learning curve is definitely the biggest problem I see. Most people have been developing web apps for years, which consist of a server component processing requests and sending HTML back to the browser.

This is a completely different paradigm which resembles network programming (client/server) a lot, and that's not something many have done before. It has many of the same problems network programming has, and having a background in it surely helps.

Thomas Davis: The only major pitfall to really worry about is not having server-generated content available at the URL. This of course stops search engines from being able to directorize your website and you will also encounter problems with sharing your page. A lot of social networks now try to parse your website when users share so that they can bring down extra information. If you don't have content generated at server time, these share attempts will fail. Though this problem has been solved many times and I recommend prerender.io as nice open source solution, I have also written SEO server libraries to mitigate this problem. Though the search-engine giants should really take it into their own hands to render your single-page applications for content. Some people speculate the entire Chromium project is an attempt by Google to be able to load and execute all pages so that they can tie it into Googlebot and directorize all content rendered by JavaScript.

What would be the fastest and most efficient learning path for someone that wants to start working with the framework? Any resources you'd like to recommend?

Igor Minar: There are several good books on AngularJS, newsletters like ng-newsletter.com, and many great podcasts at egghead.io.

Julien Knebel: The official guides at emberjs.com are tremendous; every single one of them targets a specific feature of the framework. Otherwise, I wrote an in-depth introduction for beginners for

THE FUNDAMENTAL PROBLEM I LOOK FOR ANY FRONT-END FRAMEWORK TO SOLVE IS TO GIVE ME SOMETHING WITH SUFFICIENT BONES, MUSCLE, AND CONNECTIVE TISSUE TO HELP ME BUILD A COMPLEX UI

Smashing Magazine (when Ember.js 1.0 came out) that, in my humble opinion, is still very relevant.

Brad Dunbar: It's maybe not the coolest thing to point out, but I always recommend reading the source. That's where the action is and if it's a mess, you'll know it. You'll become very familiar with your choice of MVC framework so if you can't stomach the source you may as well hang it up.

John Munsch: If the person in question is already familiar with JavaScript, I would recommend picking a small project (four or fewer distinct views making up a web app) and implementing that entire project using the framework in question. Building something, even something very simple, all the way to completion and actually deploying it for use is very different from playing with a framework. You are forced to solve some specific problems and really start to understand some things that are otherwise easy to skip because they may not be the fun parts of learning it.

Julio Cesar Ody: Make as many mistakes as you can in pet projects, but be extra mindful about them. By that, I mean it's futile to try and nail everything in the best possible way right off the bat, and it won't help you get there faster.

Then refactor your own code over and over, keeping in mind that you're looking for a scenario where you're putting a bunch of components to work together to become a full application, and you need to keep them as independent as possible.

I've written a free booklet on this, with an emphasis on Backbone.js, which of course I'll recommend. ▶

Thomas Davis: It depends on your learning style. I generally prefer to just jump in and start hacking away. I have received amazing feedback on my video tutorial for Backbone.js, which can be found on YouTube.

Now that MVC frameworks have become mainstream, how do you think they need evolve in order to better fulfill developer needs? For example, what features do you think they're still missing?

Igor Minar: Mobile, testability, mobile.

Julien Knebel: Real-time data synchronized between the back end and multiple clients would be the next logical step. Meteor seems to go this way so I'll surely give it a try soon.

Brad Dunbar: I'm not sure about this one. I tend to submit patches for things I think Backbone.js needs and not all of them are good. :)

John Munsch: The easiest answer is that something like the usage numbers at Angular Modules tells you some of the solutions that developers end up having to seek out over and over – anything that ties the front-end framework to HTML/CSS frameworks (like Bootstrap), file upload, internationalization and localization, etc. If hundreds of developers have had to go find it and then took the time to mark on a website that they use it, you can bet that thousands more also use it but never bothered to share that fact. That's the easy answer, but I'd actually plug two things I don't think get the love they ought to: validation and no back end.

Validation

Client-side validation is usually wanted and for some pages can be very complex. We have pages where many dates need to be tested against each other to make sure they occur in the correct sequence, and many other variables have various constraints as well. Then we send them up via an API call to the server, which turns around and tries to do the same stringent validation because you can never rely on validation client-side. Our back end is not written in JavaScript so we can't use a common library of code to do the validation and we get to build two sets of the same code in different languages and give ourselves twice the likelihood of making a mistake in some edge case.

Why not let me describe the data with a JSON Schema and then make it easy for me to use that for validation on both sides of the API call? I would like to see that made easy.

No Back End

There was some buzz about this last year as a term of art (for example, nobackend.org) but that's kind of died down. But the idea itself still seems to be going strong with Firebase, Hoodie, Backendless, etc. I think it's a great way for some developers only familiar with front-end work to have a way to do a complete app without needing help from someone to do the back end. But even for someone like me who could comfortably build the front and back-end of an application, it provides an easy way to prototype an idea or get to an MVP.

Julio Cesar Ody: It's hard to say. I think a lot of ideas are borne out of misconceptions about modularity, for one, so the "more features" mindset is one I've historically fought in the past.

But I think React has the right idea about it, in that it gives you a component-driven approach to building apps, and it's hard to do anything terribly wrong if you just follow the examples. It also abstracts some hard problems such as DOM performance and it pretty much gives you that for free.

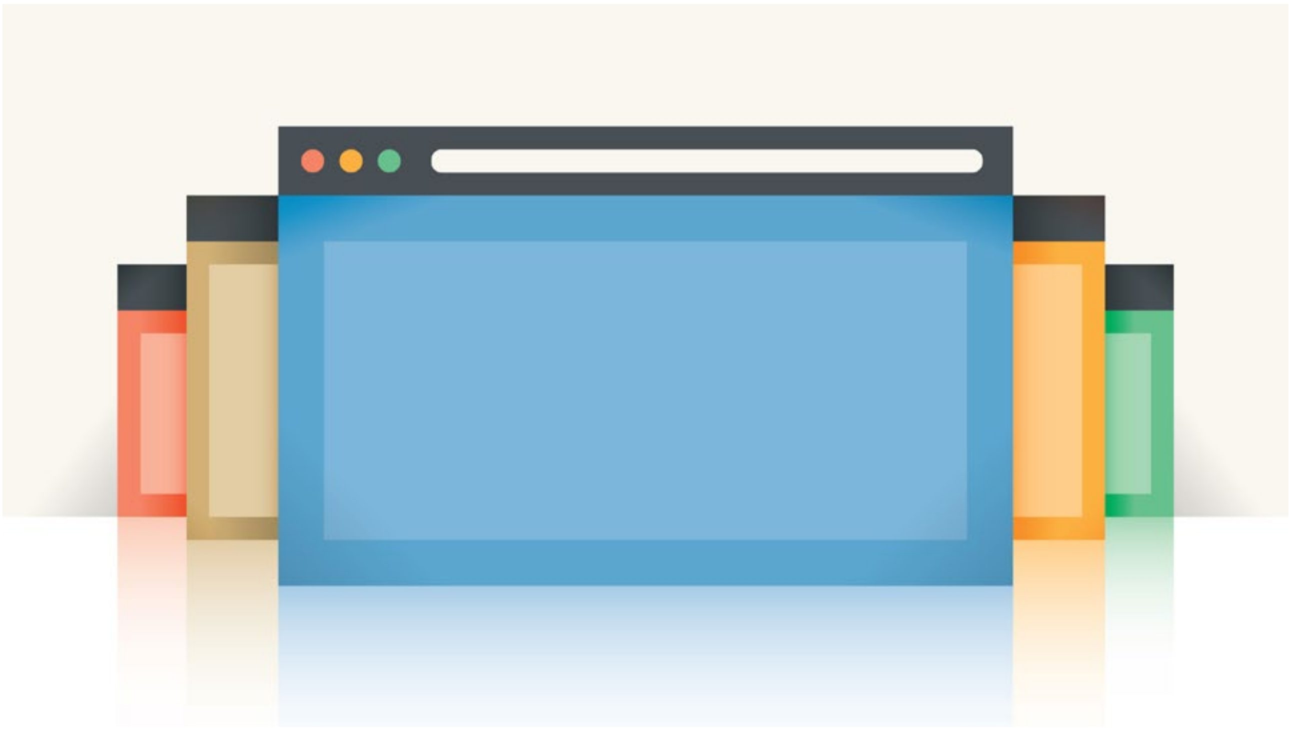
That's the kind of problem no one wants to spend any time solving in the course of building an application, so that's a step in the right direction.

Thomas Davis: Client-side tooling and MVCs are making fast progress in the right direction. I actually believe that server APIs are lagging behind at this point. There is no real convention that people are following when it comes to building RESTful APIs, which makes it difficult for client-side models and collections to bring down data efficiently. Error logging on the client side still needs some work but attempts such as Track:js are making progress there. How events are handled on the client side could also use some work. ■

READ ONLINE ON InfoQ

Q&A

Matthew Carver on *The Responsive Web*



by David Iffland

Matthew Carver is a front-end developer and web designer with real-world experience in responsive design for clients like American Airlines, the Dallas Morning News, and Chobani Yogurt.

In his book The Responsive Web, Matthew Carver provides an accessible introduction to modern web design and the importance of responsive design.

The book is divided into three sections. The first section provides a brief introduction to responsive design, why it exists, and some of the features of modern web browsers that enable the responsive Web to exist. It also makes the case for designing mobile-first: designing sites for a mobile-device breakpoint before moving on to a desktop design.

The second section goes into the how of designing for a responsive web. Matthew offers information on how to present ideas to clients before writing any code. Using navigation as a talking point, he introduces design patterns and the thoughts behind building responsively. The book goes on to cover some of the fundamentals of responsive design and techniques for presenting information, such as web typography.

In the final section, Matthew introduces more advanced techniques as effective solutions to problems that various browsers present. The book wraps up with practical advice on testing a design and optimizing it for performance.

InfoQ spoke with Matthew about his book and the challenges facing the modern web developer.

Modern developers face incredible challenges with myriad browsers and devices. What advice can you give to developers struggling to keep up?

First of all, I would say keep at it. I think the state of "struggling to keep up" is common among passionate developers. Our industry moves way too fast for anyone to truly live on the bleeding edge without having a very focused, niche practice. If you feel like you're struggling to keep up, then you're likely doing your job pretty well. ▶

I THINK THE STATE OF "STRUGGLING TO KEEP UP" IS COMMON AMONG PASSIONATE DEVELOPERS. OUR INDUSTRY MOVES WAY TOO FAST FOR ANYONE TO TRULY LIVE ON THE BLEEDING EDGE WITHOUT HAVING A VERY FOCUSED, NICHE PRACTICE.

Secondly, there's so many great ways to emulate devices and browsers. It would be great to have a fully stocked device lab with all the up-to-date tablets and mobile phones, but it's simply not a reality for a lot of developers. Luckily, there are some alternatives.

Chrome offers in its developer tools a sort of emulator that frames the viewport relative to a device's size, so you can choose to emulate the screen size of any number of devices. It'll also spoof things like touch controls so you can get a pretty good sense of what to expect in mobile and tablet browsers right in Chrome. Installing the Android SDK will install the official emulator, and Xcode comes with an emulator for iOS devices.

What are your thoughts on frameworks such as Foundation or Bootstrap? Where do they fit into the developer's toolbox?

I was pretty soft on them previously. In The Responsive Web, I discuss Foundation in detail and offer it as a great prototyping tool, but recently I've been experimenting with it in production environments. The most common argument against these frameworks is that they can be bloated or restrictive to the design. I feel like they represent a real need in the process. though.

Site builds are getting more and more complex as we stretch our legs and start using the Web to solve more and more real-world problems, so when you set out with a site build, sometimes you're not sure where you'll end up. Having a framework like Foundation in place offers a prefabricated solution that frees you up from solving an immediate problem like "What do buttons look like on the site" and lets you move into more complex problems such as "Is there a better way to implement this button?" I still end up customizing 70-80% of the framework, but using Foundation or Bootstrap as a starting point saves a ton of time.

What do you say to critics who say that responsive design causes bandwidth and memory issues on mobile?

I think that's a valid criticism in a sense. The simple fact of the matter is that in web development, there has never been a large-scale solution offered that hasn't in one way or another been controversial. Has there ever been a solution to a web-development problem that wasn't followed by a swarm of critics? Look at Swift right now. Apple has more cash reserves than some countries and has spent years building a language to improve iOS development and immediately there's criticism over how well it compiles. Face it: as developers, we're a critical lot, and if you want to know how failing to embrace change works out online, go talk to a Flash developer.

Bandwidth and memory exist in a budget and in order to accomplish tasks you must spend that budget. Developers might overspend in those budgets for myriad reasons but it's not a valid reason to dismiss responsive design as a whole. That's just silly. There's this old saying, "A shoddy carpenter blames his tools." Responsive design is a tool to solving the problem of device parity on the Web. Device fragmentation is a reality on the Web and just because responsive design isn't perfect doesn't mean it's worth abandoning.

Integrated browser tools (like F12, etc.) continue to advance in functionality and complexity. What improvement excites you the most about these tools? What parts are most useful to responsive devs?

I think the movement to build tools is pretty incredible, still. I know it's a little old hat, but Gulp and Grunt have made the development process so much more efficient, by leaps and bounds. Compass's ability to generate sprites on the fly is incredible and tools like LiveReload make the whole front-end development process faster.

How will we see the responsive Web change in the future?

That's a hard question. In the appendix of The Responsive Web, I have a chapter on what I call "context-aware design". This idea, that an interface should adapt deeply to a user's environment and patterns, is something that I found was already being discussed at Code and Theory as a "responsive philosophy". Just that fact that this same concept was cropping up independently of each of us both looking at the same problem proves to me that there's something there.

I think we have all the tools in place to offer a deeper connection to UI for web users that incorporates their personal preference, environment, time, and personality into interface design.

What are your thoughts on tools like Macaw and Brackets for designing and coding?

They are great. I don't know that I would use either in production at the moment, but any tool that helps expose designers to the fluctuations in space that are common in responsive design is great. The ability to manipulate and move the drawing space in a design helps designers articulate their ideas closer to the literal browser. I find more and more arguments towards designers using a more fragmented tool set, instead of the one application to rule them all approach of Photoshop.

What was your main motivation for writing the book?

I wanted to create something to resolve what I saw as the biggest issue in modern web design: collaboration between disciplines. My hope was to write something that a designer could read and come away from empathizing with the challenges of a developer, and that a developer could read and empathize with the challenges of design. I care deeply about the Web and I believe passionately that we can use the Web to make lives better but that can only happen if we first develop systems to solve problems and make these solutions usable for average people.

A great developer might can come up with a great new way to search for homes to buy that cuts costs dramatically but if it relies entirely on a command-line interface, does it do anyone any good? Alternatively, a designer might come up with a beautiful interface pattern but if the software under it doesn't work, who cares? We need people with these skills to communicate for things to move forward and I hope my book can help that to some degree.

How do you think responsive design has impacted the work done by today's web developers?

I think it's inside of every function on the process. You can't ignore users' device preferences and I think responsive design has found its place in the development workflow. To me, it's the default process and I find myself having to justify not including responsive layouts by default.

Note: Save 40% on The Responsive Web with discount code **info40rwd** at manning.com. The code is active and will not expire. ■

QCon
NEW YORK
Jun 8-12, 2015
4TH INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE
SAVE $100
WHEN YOU REGISTER WITH PROMO CODE
JAVASCRIPT

TOWARDS A RESOLUTION Independent Web with SVG

Angelos Chaidas is currently the senior front-end developer at Adzuna, the international job search engine. He started his career in 2000 as a designer and full-stack PHP developer but has focused on front-end and JavaScript development for the last eight years. He loves mobile UX and UI design, is passionate about web optimisation and has spoken at local JavaScript events. You'll find his random Twitter thoughts at @chaidas.

There are advantages to using Scalable Vector Graphics (SVG) as the preferred format for the graphic assets of any web or mobile web project.

The aim here is not to deter designers and developers from the proven method of using raster formats (PNG/JPEG) but rather to showcase how using SVG can enhance the workflow of the two main lifecycles of any web project: the design and the development stages.

Design

Ideal for resolution-independent user-interface icons

At the time of this writing (October 2014), flat design is an unavoidable meme. Microsoft is applying its modern design principles to all software as well as mobile-device user interfaces. With the advent of iOS 7, Apple masterfully substituted their clever skeuomorphic principles in favour of clear guidelines for flat design. Not to be left behind, Google is pushing for its "material design" visual language to be used everywhere, from Android apps to websites. (Image 1)

With complex, pseudo-3-D UI backgrounds evolving into primitive colours and sculpted buttons turning flat - both approaches easily implemented with HTML elements styled with CSS - the focus of UI design is moving towards typography, layout, and icons.

The main advantage of SVG, rightfully advertised all over the Web, is its scalable nature. When preparing an icon, a designer can stop worrying about how it will look in different dimensions or on mobile devices with different pixel densities and focus solely on making the asset look great.
With SVG, there is no need to export two different raster assets for high-resolution and lesser-resolution displays. As a matter of fact, there's no need to worry at all about the pixel density of any device, especially

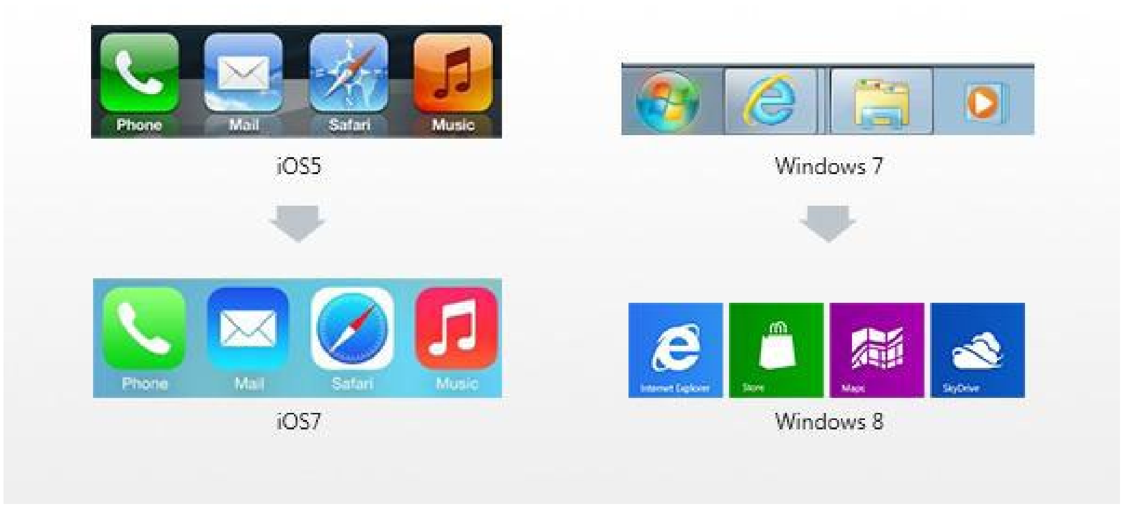
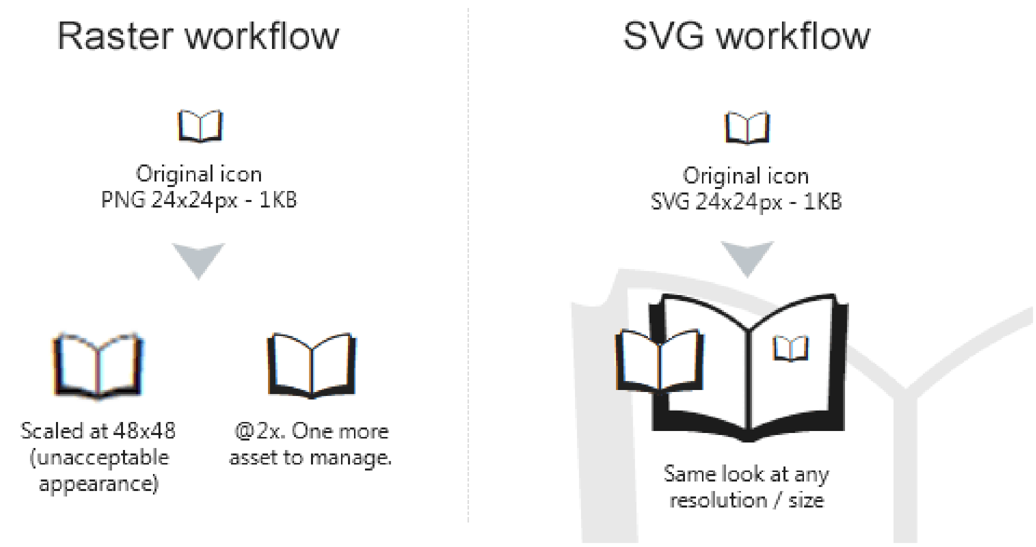


Image 1 / The flattening of design: glows, drop shadows, and pseudo-3D skeuomorphism are evolving into simple shapes.



Raster workflow
Original icon
PNG 24x24px - 1KB
Scaled at 48x48 (unacceptable appearance)
@2x. One more asset to manage.
SVG workflow
Original icon
SVG 24x24px - 1KB
Same look at any resolution / size

Image 2 / The SVG asset can be scaled at whim by the developer with no loss of quality. There's no need to export @2x (or @3x) versions of an asset, resulting in one less step in the designer's workflow.

given the (sometimes absurd) non-standardised nature of it, but focus rather on the artwork itself. Once an icon is ready, the designer exports a single SVG file - directly from Illustrator - that can be resized at whim by the developer with no loss of quality. (Image 2)

Browser scaling is getting better and better, even on small dimensions

There's a common design challenge for the slightly-more-obsessive-compulsive-than-normal designer: in Photoshop, an icon might look great in the finger-friendly dimensions of 44x44 pixels, but resize it to 24x24 or 16x16 and the bicubic interpolation introduces anti-aliasing artefacts which might result in a blurry image. Even with clever algorithms such as the sharp bicubic interpolation available in the latest versions of Photoshop, many designers end up drawing from scratch the smaller image assets (interface icons, favicons, etc.) to achieve clarity.

With SVG, the above scenario is partially mitigated by the fact that browsers scale (and consequently rasterize) SVG assets very well. This is especially true for higher display densities such as the latest generation of mobile devices. (Image 3)

For optimal results in these dimensions, a designer can easily package all SVG assets into an icon font (see below for more details) and thus leverage the sub-pixel hinting capabilities of the various operating systems, resulting in icons sized at 12 or 14 or 16 pixels that look crystal clear and razor sharp, even on old IE browsers.

SVG elements can be crafted to reduce complexity and file size

Optimizing a JPEG image is a one-way process of reducing quality while trying not to lose too much ▶

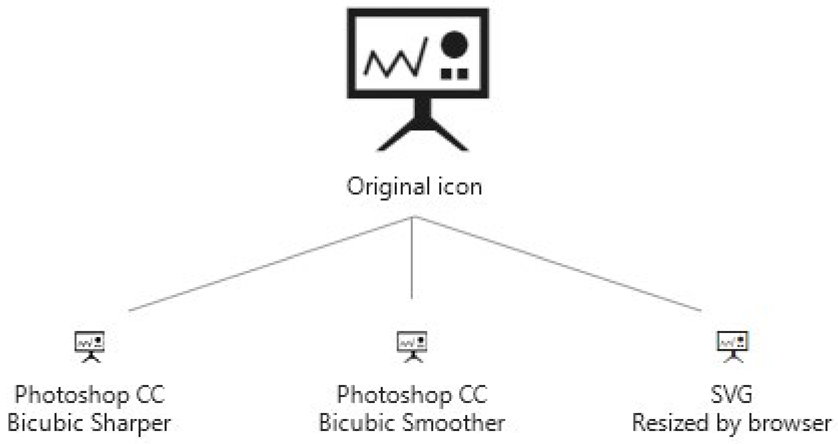


Image 3 / Browser scaling is getting better. In certain cases, the browser scaling ends up looking even better than the advanced interpolation methods of Photoshop.

information. For PNG and GIF assets, the designer has slightly more control due to the ability to specify a restricted palette and thus reduce the information encoded in the file.

With SVG, the meticulous designer can opt to use primitive shapes, reduce the number of vertices in the shape elements, and have text embedded inside the SVG instead of converted to outlines, all of which can result in a less complex and smaller SVG file. (Image 4)

For even more optimization, a designer can remove extraneous SVG properties such as comments, IDs, or redundant grouping tags (<g>) by directly editing the SVG code.

Better open-source tools

Instead of Photoshop (and perhaps Fireworks or Sketch), the open-source world offers the Gimp editor. Gimp is a valuable tool with a vibrant community, yet it is inferior to any of the aforementioned pieces of software in several areas. Substituting Photoshop with Gimp is not an easy task, and it usually results in a convoluted workflow.

For vector artwork however, Inkscape is a professional-grade open-source alternative to Adobe Illustrator that enables designers to create and edit vectors with a plethora of tools and procedures, similar to what they are used to with Adobe's offering. Many of the everyday actions of working with vectors can be done in Inkscape as well, such as Boolean operations to combine shapes, path simplification, dynamic offsetting of paths, editing subpaths and much more.

Developers

Resolution independence and reusability

For screens with higher pixel densities such as Retina displays and mobile devices, the need for @2x high-res assets goes away. With proper meta tags and the asset dimensions specified in the CSS file, a developer has full creative control when it comes to resizing the asset without the need to push back to the designer for variations of the same file.

There is something exhilarating in the fact that a single file (e.g. company-logo.svg) can be used in various places on a user interface, with the dimensions of each instance controlled by CSS or width/height attributes, and having the resulting assets look crystal clear irrespective of how much each is scaled (or rotated).

Animation

An SVG animated with CSS3 retains sharpness and clarity throughout the animation duration. A couple of examples are shown below but please note that the low frame rate of the animated GIF does not do justice to the GPU-accelerated smoothness of the finished effect:

But that is only the beginning. The structured XML syntax of SVG enables a developer to programmatically select and animate each individual element of an SVG file to create scripted animations, banners, ads, etc. CSS-Tricks has an excellent step-by-step tutorial for creating an animated banner in SVG format, and showcases how, when placed inline, individual elements of an SVG image can be manipulated with CSS to follow a certain script, resulting in an animated banner similar to those on the Web when Flash was the industry standard for animated advertisements.

Interactivity

The potential of SVG as a replacement to Flash does not stop with animation. SVG content can be interactive, with events such as click, mousedown, and mouseup (plus a few interesting ones such as SVGZoom and SVGResize) available to the developer

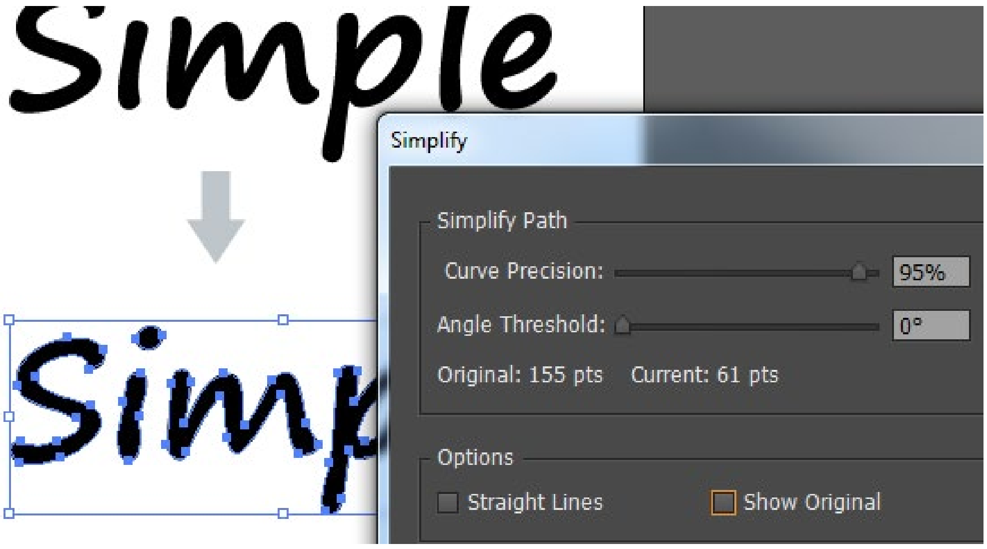


Image 4 / Illustrator's "Simplify" in action. The vertices that make up a vector path can be reduced with minimal loss of precision, resulting in a smaller SVG file.

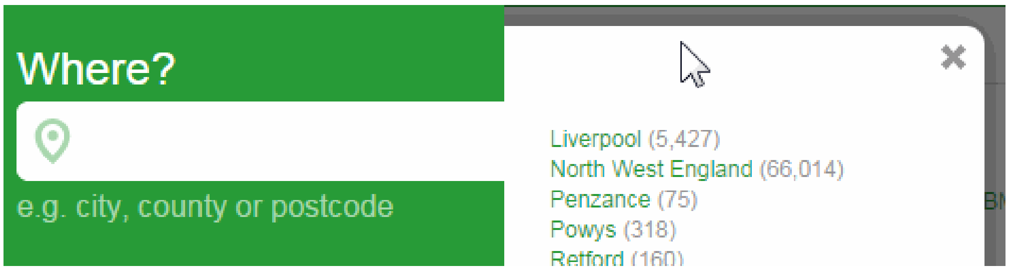


Image 5 / For a live demonstration, visit the Adzuna homepage and focus on the text fields, or click the "More" links in the bottom lists.

to programmatically listen for and respond to, using JavaScript.

This opens up possibilities such as interactive infographics and enterprise-strength charting solutions.

Going further, the developer can use helper libraries that facilitate drawing, manipulation, and interactivity, with some of the top candidates being Raphael, D3.js, and Snap.svg. With libraries like these, complex animated and/or interactive presentations that previously could only work through Flash are now possible, with several examples available for inspiration, from the New York Times's custom tree structure for political analysis of the Obama/Romney campaigns to Hamiltonian graphs, force-directed graphs, zooming interactive maps from the Argentina Census, and countless more.

Inline use

The two standard approaches to using image assets on a web project are by means of an <img> tag, and as background images for (usually block-level) elements. Since SVG is XML, a third possibility is now available to developers: inline use.

With HTML5, the <svg> element can be placed directly into the source HTML of the page. The advantage here is that the <svg> element, as well as its children elements, can be controlled with CSS.

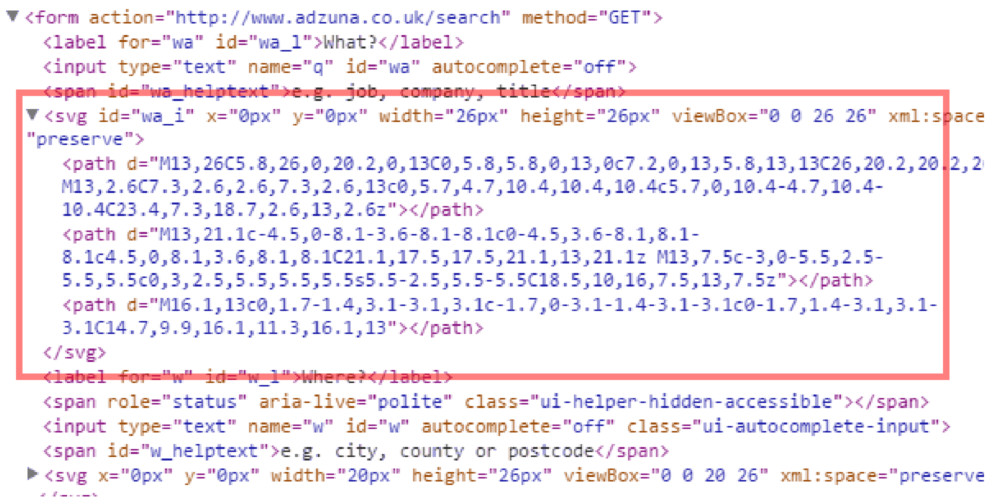
Apart from size and position, designers can also manipulate fill and stroke colours, and can even create an animation. Furthermore, certain SVG-only attributes (such as stroke-dasharray and stroke-dashoffset) can be manipulated from within the CSS file to result in interesting possibilities for animation such as this line animation.

More ways to sprite

With raster assets, a classic optimisation approach for reducing HTTP requests is to combine several images into a single sprite (usually PNG) and use the background CSS property to style several different HTML elements with that one sprite.

With SVG assets, two extra possibilities become available to developers: inline grouping and packaging in an icon font.

With inline placement, the ability to control the display of sub-elements with CSS enables a developer to create an SVG "bucket" of assets, with each group being an element such as an icon. The developer can assign IDs to individual groups, and by changing ▶



```
▼<form action="http://www.adzuna.co.uk/search" method="GET">
   <label for="wa" id="wa_l">What?</label>
   <input type="text" name="q" id="wa" autocomplete="off">
   <span id="wa_helptext">e.g. job, company, title</span>
  ▼<svg id="wa_i" x="0px" y="0px" width="26px" height="26px" viewBox="0 0 26 26" xml:space
  "preserve">
     <path d="M13,26C5.8,26,0,20.2,0,13C0,5.8,5.8,0,13,0c7.2,0,13,5.8,13,13C26,20.2,20.2,2
     M13,2.6C7.3,2.6,2.6,7.3,2.6,13c0,5.7,4.7,10.4,10.4,10.4c5.7,0,10.4-4.7,10.4-
     10.4C23.4,7.3,18.7,2.6,13,2.6z"></path>
     <path d="M13,21.1c-4.5,0-8.1-3.6-8.1-8.1c0-4.5,3.6-8.1,8.1-
     8.1c4.5,0,8.1,3.6,8.1,8.1C21.1,17.5,17.5,21.1,13,21.1z M13,7.5c-3,0-5.5,2.5-
     5.5,5.5c0,3,2.5,5.5,5.5,5.5s5.5-2.5,5.5-5.5C18.5,10,16,7.5,13,7.5z"></path>
     <path d="M16.1,13c0,1.7-1.4,3.1-3.1,3.1c-1.7,0-3.1-1.4-3.1-3.1c0-1.7,1.4-3.1,3.1-
     3.1C14.7,9.9,16.1,11.3,16.1,13"></path>
   </svg>
   <label for="w" id="w_l">Where?</label>
   <span role="status" aria-live="polite" class="ui-helper-hidden-accessible"></span>
   <input type="text" name="w" id="w" autocomplete="off" class="ui-autocomplete-input">
   <span id="w_helptext">e.g. city, county or postcode</span>
  ▶<svg x="0px" y="0px" width="20px" height="26px" viewBox="0 0 20 26" xml:space="preserve
```

Image 6 / SVG code can be placed directly into the HTML. Older browsers will simply ignore it and newer browsers will render the vector asset, which is now reachable by CSS (#wa_i) and JavaScript (document.getElementById('wa_i')).

the display property of each group chooses what to hide and what to show. This technique works really well with interface elements that have the same dimensions, such as the icons for a user interface.

A developer also has the option to package several SVG assets together in an icon font. Browser support is excellent (even IE6) and the sub-pixel hinting engines of modern browsers make even small sizes look crystal clear and sharp. Even better, multiple online icon-font generators make the (somewhat annoying) packaging process a breeze.

Comparing versions of the same asset

The SVG format, by virtue of being essentially a text file, presents the developer with the interesting possibility of not only comparing an asset visually, but also doing a file diff to establish which parts of the SVG have changed.

In the case of large and complex SVG files, such as an infographic, text comparison is great in establishing what parts of the asset have changed in a newer version.

Further reading

This author's personal preference for development is Chris Coyier's (of CSS-Tricks fame) Compendium of SVG Information as well as monitoring any SVG-related articles that the brilliant Sara Soueidan comes up with. The compendium, a huge list of links to SVG resources split by logical sections, is the definitive starting point for anyone interested in SVG.

For designers, Todd Parker's massive "Leaving Pixels Behind" presentations are the best possible introduction, packed with animated GIFs that showcase the workflow from Illustrator to SVG.

Falling back for older browsers

As of October 2013, Internet Explorer 8 is still, unfortunately, something developers need to address. Combining this requirement with the fact that pre-v3 Android Browsers lack support for SVG means that a fallback solution must be implemented.

For developers, Modernizr is the tool of choice. Including the Modernizr library as an external <script> in the <head> of any web project will add the appropriate classes to the <html> element of said document on page load. It is then a matter of adding a few extra definitions to the CSS to replace background SVG images with raster fallbacks or, in the case of inline SVGs or SVGs placed in <img> tags, showing helper tags that contain the raster fallbacks and are by default hidden.

The challenge here is to not have to push back to the designer to export fallback assets since this invalidates their "export only one SVG asset" advantage mentioned above.

Fortunately, automation tools such as Grunt and specifically Filament Group's Grunticon are here to help. In short, Grunticon operates on a folder of SVG assets and outputs a list of fallback PNG files along with a fallback CSS file that references these PNG images. For command-line gurus, Inkscape can be used as part of a shell script to also convert SVG files to any format.

Summary

The advantages of using vector graphics on the Web are now more numerous than the disadvantages. With excellent browser support and automated fallback solutions to support older browsers, it is this author's belief that future-proofing a UI with resolution-independent vector graphics is the way to go forward. ■

React in Real Life at Codecademy

Bonnie Eisenman is a software engineer at Codecademy.com. A recent Princeton CS graduate, she also has a bit of a hardware habit and enjoys working with Arduinos and musical programming in her spare time. Find her as @brindelle on Twitter.

In August 2014, Codecademy decided to adopt React, Facebook's library for writing JavaScript UIs, as part of an overhaul of our learning environment. Initially, we ran into issues finding examples of how React could apply to an application as complex as ours, given that most tutorials focus on the specifics of toy-sized demo apps as opposed to more general discussions of the problems specific to larger applications.

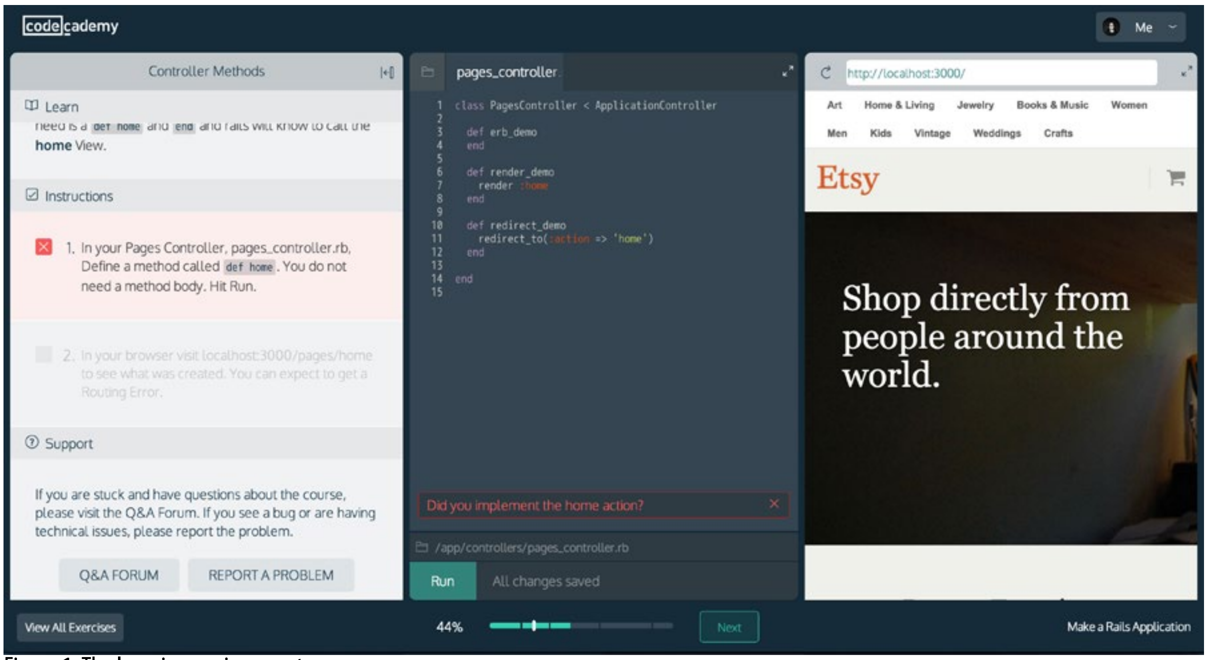
What is React?

React is a library for building user interfaces with JavaScript. Instead of the usual approach to writing user interfaces, React treats each UI element as a contained state machine. It is not a framework in the sense of something like AngularJS. Though Facebook sometimes describes React as "the V in MVC," I find this description less helpful, since React applications needn't abide by the MVC model. React helps you build fast user interfaces that can handle complex interactions without a lot of ugly code.

If you're going to be working in React, you should be aware of these features:

React handles the DOM for you.
DOM manipulation is expensive. React's appeal comes largely from the way it solves this problem. React minimizes the amount of DOM manipulation by maintaining its own virtual DOM and only re-rendering when necessary, a feat enabled by its high-performance diff implementation.

This means that you will rarely touch the DOM directly; instead, React handles DOM manipulation on your behalf. This feature is the basis for much of React's design. If you feel like you're abusing the React API or attempting to hack your way around



need is a `def home` and `end` and rails will know to call the **home** View.

1. In your Pages Controller, pages_controller.rb, Define a method called `def home`. You do not need a method body. Hit Run.

2. In your browser visit localhost:3000/pages/home to see what was created. You can expect to get a Routing Error.

If you are stuck and have questions about the course, please visit the Q&A Forum. If you see a bug or are having technical issues, please report the problem.

```
class PagesController < ApplicationController

  def erb_demo
  end

  def render_demo
    render :home
  end

  def redirect_demo
    redirect_to(:action => 'home')
  end

end
```

Did you implement the home action?

/app/controllers/pages_controller.rb

All changes saved

Shop directly from people around the world.

Figure 1: The learning environment.

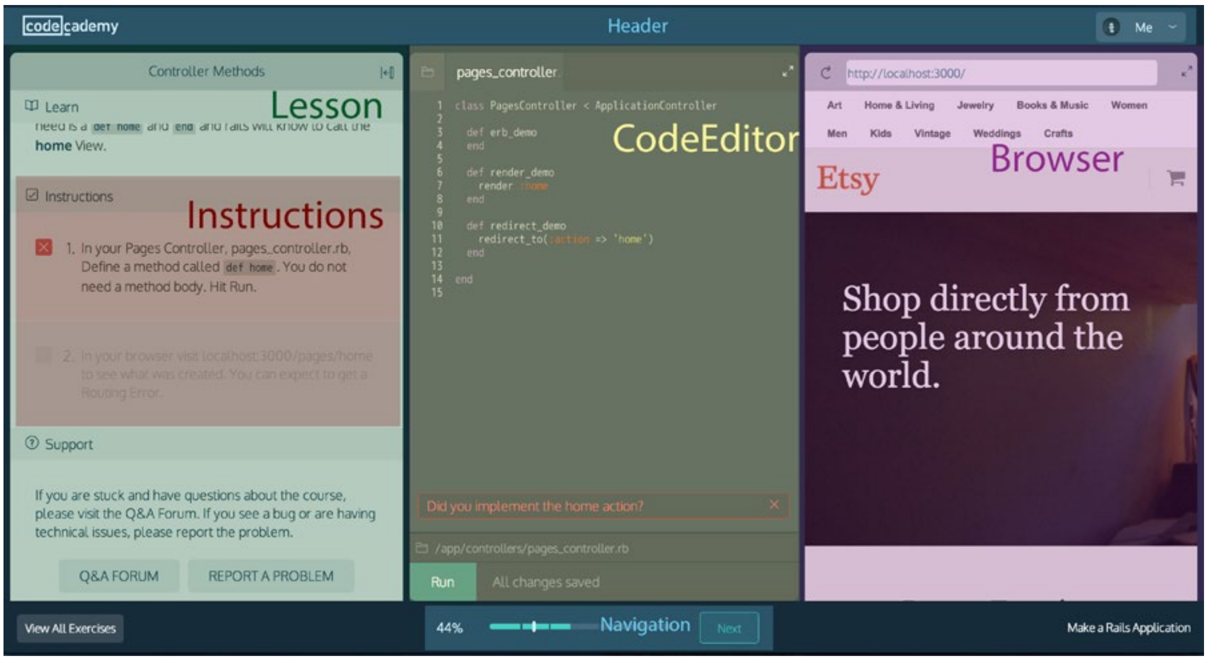


Figure 2: The learning environment and related components.

it, chances are you're interfering with React's understanding of the DOM.

This feature also enables built-in server-side rendering using Node.js, which allows you to easily serve SEO-friendly webpages.

React thinks declaratively, in Components.
In React, everything must subclass the Component class. Components have properties (determined by their parent) and state (which they can change themselves, perhaps based on user actions). Components should render and behave based solely on their state and properties; components are state machines. This model encourages building modular UIs and, in practice, makes it easier to work with and reason about your UI.

React marries markup to JavaScript.
Though it may feel strange to write HTML in your JavaScript, in React it's the natural way to do things. Using JSX – plain JavaScript mixed with HTML – is optional but I highly recommend it. React argues, and I agree, that since your markup is tightly coupled to the JavaScript that controls it, they may as well live in the same file. ►

Information flows in one direction.
This is more of a general React pattern than a strict rule. Information flow *tends* to be unidirectional in React. We'll revisit this pattern later as we begin considering how information flow needs to be handled in larger applications.

Anatomy of a React application

To make these principles more concrete, let's look at how React works for the Codecademy learning environment. (Figure 1)

As you can see from the screenshots, the main learning environment consists of many different UI elements. Some, like the header, menu, and navigation, are present at all times. However, depending on the exercise, some components appear and disappear – the web browser, command line, and code editor can be mixed and matched depending on the lesson.

The logical solution is to create React components for the various pieces. In the screenshot below, I've highlighted our main React components: (Figure 2)

Each component also may contain a number of child components; for instance, the lesson panel on the left is actually composed of a number of components: (Figure 3)

In this case, we use React to determine what appears in the lesson panel. For example:

- Only show the "Report a Problem" button if the user is signed in.
- Only render the Instructions section if this exercise has tests.

Furthermore, React handles information flow between this component and others. There's a parent component for the entire learning environment, which keeps track of state such as what exercise the user is on. This parent component dictates how its children should render by setting their *props (properties)*.

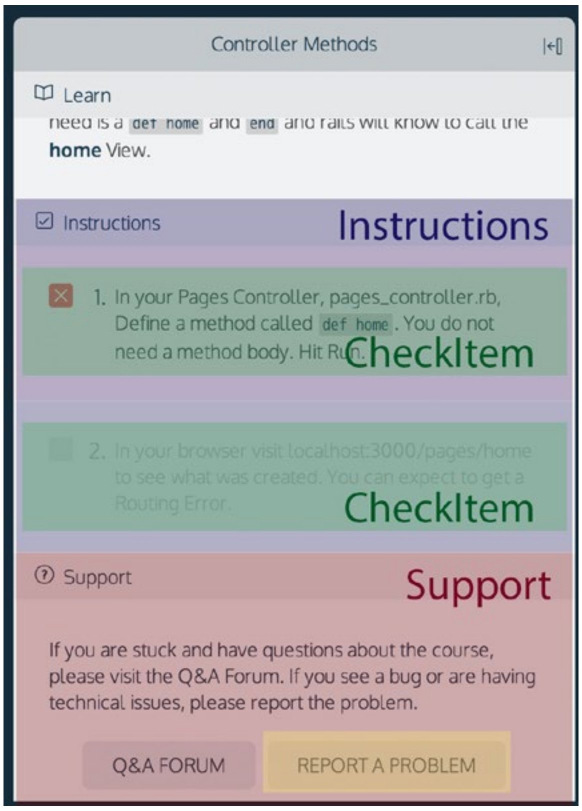
Now let's look at an example of component communication, using the following components in this much-simplified component tree:

- LearningEnvironment
- CodeEditor
- RunButton
- ErrorDisplayer
- Navigation

(Figure 4)

How do we handle the workflow of users trying to run their code? We want to run tests against their code and either display an error message or allow them to proceed. Here's one possible flow:

- When the user clicks on the RunButton, it informs its parent, the CodeEditor, of the action via callback.


Controller Methods

Learn

need is a def home and end and rails will know to call the home View.

Instructions

1. In your Pages Controller, pages_controller.rb, Define a method called def home. You do not need a method body. Hit Run.

2. In your browser visit localhost:3000/pages/home to see what was created. You can expect to get a Routing Error.

Support

If you are stuck and have questions about the course, please visit the Q&A Forum. If you see a bug or are having technical issues, please report the problem.

Figure 3: The sub-components that comprise the Lesson component

- The CodeEditor then informs its parent, the Learning Environment, through another callback, passing along the user's current code.
- The LearningEnvironment runs tests against the user's code.

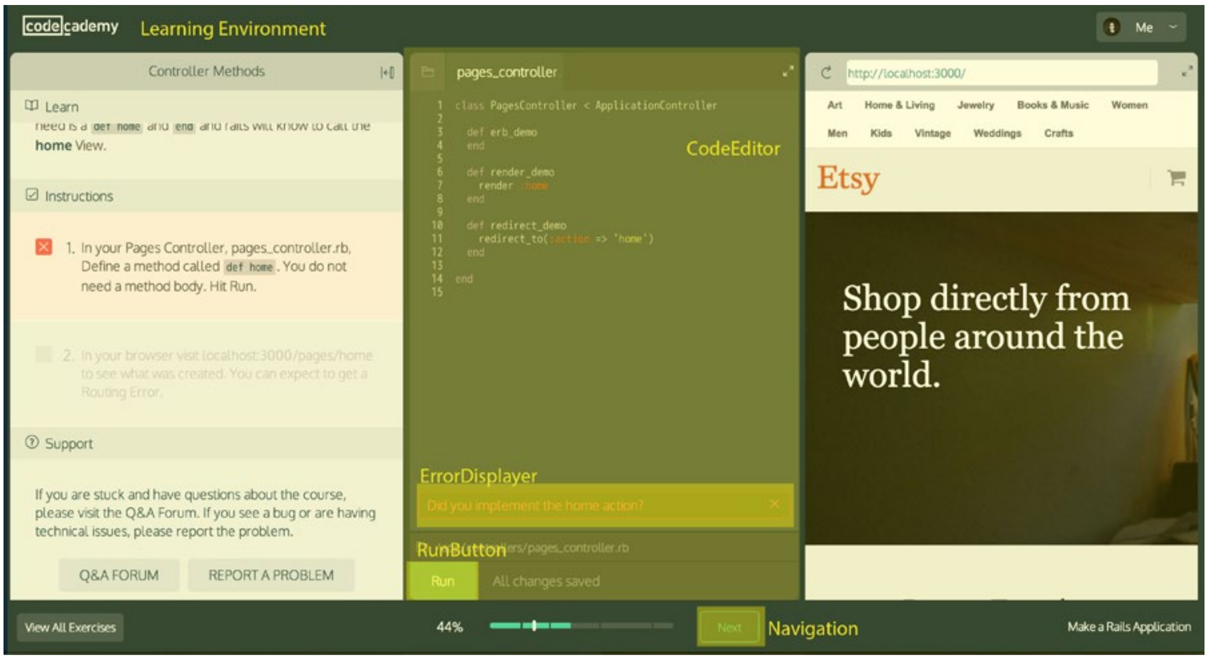
Based on the result:

- The LearningEnvironment sets the errorMessage prop on the CodeEditor, which then sets the errorMessage prop on its child, the ErrorDisplayer.
- If the user has passed all tests for this exercise, the LearningEnvironment sets the progress prop on the Navigation component.

The updates to the UI can be performed with a single function call. If our components are declared like this in the LearningEnvironment's render method (again, with much simplification):

```
render: function() {
  return(
    <div>
      <CodeEditor
        error={this.state.error}
      />
     <Navigation
       mayProceed={this.state.mayProceed}
     />
   </div>);
}
```

Remember, React mixes JavaScript with HTML. In this case, the render method is defining



need is a `def home` and `end` and rails will know to call the **home** View.

1. In your Pages Controller, pages_controller.rb, Define a method called `def home`. You do not need a method body. Hit Run.

2. In your browser visit localhost:3000/pages/home to see what was created. You can expect to get a Routing Error.

If you are stuck and have questions about the course, please visit the Q&A Forum. If you see a bug or are having technical issues, please report the problem.

```
class PagesController < ApplicationController

  def erb_demo
  end

  def render_demo
    render :home
  end

  def redirect_demo
    redirect_to(:action => 'home')
  end

end
```

Did you implement the home action?

ers/pages_controller.rb

Figure 4: Some of the components involved in code submission.

a LearningEnvironment as something that contains both a CodeEditor and a Navigation component.

We can update the LearningEnvironment's state, which will trigger a re-render and update child components as necessary:

```
handleTestResult:
    function(currentIndex, passing,
    error) {
  this.setState({
    error: error,
    mayProceed: passing &&
    currentIndex === this.state.
    currentExercise.tests.length-1
  });
}
```

That's it. React updates the UI for us gracefully and simply.

Considerations for larger applications

Information flow

As I noted earlier, React doesn't necessarily adhere to a MVC model; in fact, you're free to handle information flow however you'd like, but you'll need a conscious strategy for information. Do you want to pass down chains of props to components that don't actually need them, just to reach a great-great-great-great-grandchild? If that leaf node accepts user input, how does it alert its great-great-great-grandparent to the change?

In larger applications, this can become frustrating. Even in the simple example above, how should the Run button communicate user actions to the LearningEnvironment? We need to pass callbacks around, but it's hard to write truly modular, reusable components that way.

Codecademy's solution has been to generate communication adapters that manage information flow between specific components. Instead of passing callbacks, high-level components such as the CodeEditor also receive an Adapter, which provides a single interface for important communication tasks. For example, when the CodeEditor is present, the LearningEnvironment will generate an Adapter which can emit and process events related to code submission.

This approach isn't without its pitfalls, and I spoke about this at greater length at React.js Conf. The main takeaway that I've had is that, regardless of how you handle information flow up the component tree, your team ought to agree on a coherent strategy.

Integration

React is easy to get started with, but it does require some tooling to use effectively in your workflow. For example, we use:

- a script that watches for local changes to .jsx files, and recompiles them as necessary;
- a separate Node.js server that handles server-side rendering; and
- developer tools to auto-generate new component files as needed. ►

None of these are too complicated. Gulp is a great choice for the .jsx watcher, though we wrote ours in Go. To generate new component files, we use a simple bash script, which also enforces our naming conventions. If you're using a Node.js server for server-side rendering, beware: it can be difficult to force RequireJS to pick up changes to your React code. We have our watcher restart the node server as necessary.

Why React?

When we were overhauling our learning environment, we had to determine what tools or frameworks to make use of. We eventually chose React and are very happy with our choice. (You can find more detail about how we chose a JavaScript framework, or lack thereof, in this talk: https://www.youtube.com/watch?v=U5yjPG5mHZ8)

Here are some of the aspects that we appreciate about React:

It's battle-tested.
React is in production use at Facebook and Instagram, so we can be confident in its performance and reliability. So far, it's served us well, and we haven't experienced any significant issues.

Components are easy to think about.
Because React deals solely with individual components, which render based on their internal state, it's easy to conceptualize what ought to be happening at any given time. Your application effectively becomes a very large state machine. This means that you can test individual pieces of the UI in isolation as well as add new components without worrying about interfering with the rest of your application.

SEO is easy.
Because React has built-in support for server-side rendering, you can serve a mostly complete page to search engines, which is a huge boost to SEO with very little effort. Granted, this only works in Node.js. Since Codecademy's main app is written in Rails, we run a separate Node.js server that only handles React rendering.

React is compatible with legacy code and flexible enough for the future.
Whereas adopting an entire framework is a big commitment, you can slowly experiment with adding React to an existing codebase. Likewise, if we need to move away from React in the future, we can do so fairly easily. At Codecademy, we decided to write a new project entirely in React, to try it out and learn how best to use it; that worked out well, and now we use it for nearly all new UI elements. I'd recommend diving in, building some experiments, and then considering how React might fit in with your existing codebase.

Stop worrying about boilerplate.
Less time spent writing boilerplate code means more time solving interesting problems. From this perspective, React is concise and lightweight. Here's the minimum code required to create a new component:

```
var dummyComponent = React.createClass({
  render: function() {
    return (<div>HTML markup in JS, what fun!</div>);
  }
});
```

It's short and to the point. What's not to like?

We have a community to build with.
The React community is growing rapidly. When you encounter issues, there are plenty of people to discuss them with. And because many large companies are using React in production (Facebook, Instagram, Yahoo!, GitHub, and Netflix, to name a few), we're in good company.

In summary

React is a lightweight, powerful, battle-tested library for building user interfaces with JavaScript. It's not a full framework, but rather a powerful tool that may well change the way you approach front-end development. We've found it to be an incredibly useful tool for our front-end development, and we're very happy with our choice. For myself, at least, working in React has dramatically impacted how I think about writing user interfaces. I'm also excited to see React grow: now that Facebook is bringing React to mobile with React Native, I suspect the future is an exciting one.

If you want to get started with React, the tutorial is the logical place to begin. There are also plenty of posts that introduce key React concepts (this slide show is one of my favorites). Go ahead – dive in, try building something, and see what you think of React's approach to front-end development. I'm eager to hear what you think, and I'll be listening on Twitter as @brindelle. ■

PREVIOUS ISSUES

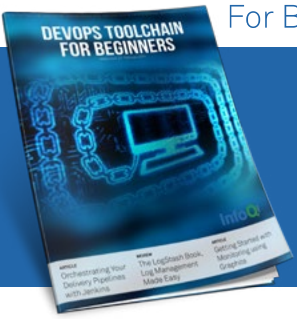
MOBILE
RECENTLY NEW TECHNOLOGY AND ALREADY COMMODITY?
eMag Issue 24 - February 2015
InfoQ
INTERVIEW Virtual Panel on App Development
ARTICLE The Swift Programming Language
INTERVIEW Behavioral Analysis for Mobile Apps

24

Mobile - Recently New Technology and Already a Commodity?

This eMag discusses some familiar and some not too familiar development approaches and hopefully will give you a helping hand while defining the technology stack for your next mobile application.

DevOps Toolchain For Beginners



23

This eMag aims at providing an overview of an example set of tools that would constitute a typical toolchain. These are popular tools today, but you should look at them as illustrations of the kind of tasks and workflows you might need to perform in your organization as you move along a DevOps path.

Web APIs: From Start to Finish

22

This eMag contains a collection of articles and interviews from late 2014 with some of the leading practictioners and theorists in the Web API field. The material here takes the reader on a journey from determining the business case for APIs to a design methodology, meeting implementation challenges, and taking the long view on maintaining public APIs on the Web over time

Continuous Delivery Stories

21

Reaping the benefits of continuous delivery is hard work! Culture, processes or technical barriers can challenge or even break such endeavors. With this eMag we wanted to share stories from leading practitioners who've been there and report from the trenches. Their examples are both inspiring and eye opening to the challenges ahead.