jRWD

a JavaScript framework for responsive web design

Contents

jRWD overview

jRWD is a lightweight framework for implementing responsive web design.

It is written in pure JavaScript. There is no jQuery involvement because jRWD can do some of the clever things that jQuery does.

jRWD consists of two small JavaScript functions, totalling 12 lines of code, and two small support functions. The entire framework contains fewer than 750 characters. jRWD doesn't require any external libraries.

setup() is the framework's primary function. It extends JavaScript's Object constructor to allow HTML elements to be switched on and off. It processes a set of browser breakpoints and sets up Javascript media queries for them. Then it adjusts the browser width to match the viewport of the viewing device.

event(), the framework's secondary function, is called when a media query event-listener fires. The function determines which breakpoint triggered the event-listener, then transfers control to a user-written JavaScript function to process the event. Flipping a smartphone or tablet between landscape and portrait modes is a typical action that will trigger a media query event-listener.

css() is a support function that gives the framework access to a web page's CSS stylesheets. Stylesheet rules can be altered in real-time, providing full, dynamic control over a web page's contents when the viewport changes size.

id(), another support function, provides access to those HTML elements that have an id associated with them, e.g., ** Hello! **.

tags(), a support function that isn't part of the framework, manages groups of items such as menus, carousels, and sprites.

jRWD completely eliminates the use of @media queries in CSS stylesheets. This has two major advantages: stylesheets become smaller and less complex; and the behaviour of web pages is more fully controlled by JavaScript, which accords with the original W3C Web Standards Model.

jRWD is open source software, available at GitHub under the MIT licence. It is compatible with all modern browsers and most mobile devices.

jRWD internals

JavaScript is an object-oriented language, but unlike other object-oriented languages it doesn't have *classes*. *Objects* and *classes* are synonymous terms.

Everything in JavaScript is an object, except **var**, though a var that is an object can be created easily: **var x = {}**. Strings are objects in JavaScript; so too are arrays, dates, and Math functions.

Inheritance is an important feature of object-oriented languages. JavaScript objects inherit properties and methods from the Object.prototype. jRWD uses prototype inheritance to give all objects, including those created in the DOM (Document Object Model) two special methods: they can be turned on and turned off as required. Every element, including images, that has an HTML id assigned to it can be turned on or turned off at any time.

setup()

The first two lines of *setup()* give every HTML element that has an id the ability to be turned on or off:

1. **Object.prototype.off = function() { this.style.display = 'none' };**
2. **Object.prototype.on = function() { this.style.display = '' };**

How does it work? If there is a **<p id='text'>** element in a web page's HTML it can be turned off with *text.off()* and on again with *text.on()*. This feature gives jRWD the ability to display or hide DOM elements, eliminating the CSS stylesheet rules that usually accomplish these tasks.

The next line of code creates 4 variables, one of which is an important array:

3. **var w, x = 1, y = [0], z = y.length;**

A typical responsive design might have an array that looks like this:

y = [0, 380, 414, 580, 720, 768, 990]

The first item in the array, *0*, is mandatory; it acts as a fence, but it is ignored by the media query event-listener mechanism. The other items in the array are browser breakpoints, and they must appear in ascending numerical order.

Each non-zero item in the array represents a breakpoint, expressed in pixels. If, for instance, the viewport width suddenly becomes greater or smaller than 380 pixels, an event-listener will fire and control will pass to a user-written JavaScript function (*function px380* in this case) to process the new viewport width. User-written JavaScript functions are described here.

```
4. for ( ; x < z; x++ ) {
5.     w = window.matchMedia ( ' (min-width:' + y[x] + 'px) ' );
6.     w.addListener (event); }
```

Lines 4, 5 and 6 create a loop that cycles through the breakpoints in the array, setting up JavaScript media queries and event-listeners on each breakpoint.

Line 5 creates a JavaScript media query (e.g., min-width: 380px) and assigns it to the *window* object using JavaScript's *match.Media* API, which works on all modern browsers and mobile devices that have JavaScript installed.

Line 6 adds a JavaScript event-listener to the media query. The JavaScript call-back function *event()* executes when the JavaScript event-listener fires. *event()* is described later in this section.

Lines 5 and 6 replace all the CSS @media queries that populate the stylesheets of responsive websites. JavaScript media queries work in much the same way that CSS @media queries do, with a minor variation that is explained here.

```
7. w = window.innerWidth;
```

Line 7 puts the viewport's current width into a worker variable.

```
 8. for ( --z; z; z-- ) {
 9.     x = y[z];
10.     if ( w <= x ) {
11.         eval ('px' +x + '(false)'); } }
```

Lines 8 - 11 set up a loop that processes items in the breakpoint array from right to left (i.e., backwards) to step the browser down to the size of the viewport. This will have no effect if the framework is running on a PC or a laptop, but it is vital for initialising the viewport on mobile devices.

While stepping the browser down to fit the viewport size, the loop executes the media queries that change the web page layout to fit a smaller form-factor.

Line 11 contains an *eval()* statement. This may trouble some framework users. For a discussion of *eval()* and how to replace it in the framework, please read this section.

Line 11 is the last line in the *setup()* function.

event()

event() is a single-line JavaScript function that is called each time a media query event-listener is triggered. *event()* establishes which media query was involved, then passes control to a specific user-written JavaScript function to process the event.

event() will call *function px380* when the breakpoint at 380 pixels is triggered.

12. eval ('px' + e.media.replace(/\D/g,'') + '(e.matches)');

In line 12 *event()* parses the JavaScript object returned to it by the media query. It extracts the event-listener function number from the object and immediately passes control to the relevant user-written JavaScript function, sending the object's *true/false* value as an input parameter.

Line 12 contains an *eval()* statement. This may trouble some framework users. For a discussion of *eval()* and how to replace it in the framework, please read this section.

Line 12 is the last line of the jRWD framework.

jRWD support functions

function css()

The *css()* function gives the framework access to a web page's CSS stylesheets, allowing the framework to alter CSS rules in real-time. This is an extremely powerful capability, putting jRWD on a par with much larger frameworks.

How does it work? A CSS rule selector is passed to the function. The function links up all of the web page's stylesheets and searches for the rule. If the rule is not found the function returns *false*; otherwise it returns a link to the rule.

Consider a CSS rule that allocates space to two elements:

#f1,#f4{width:15%;}

The two elements may require more space if the viewport size changes, which can be accomplished in JavaScript using jQuery-like notation as follows:

css('#f1, #f4').style.width = '33%';

Alternatively the link to the rule can be stored in a variable, and the variable can be manipulated in various ways:

var x = css('#f1, #f4');
x.style.width = '33%'; etc.

function id()

function id(e){ return document.getElementById(e); }

id() is a single-line JavaScript function that locates a named HTML element in the DOM and returns a link to the element. The element's id is passed to the function as an input parameter.

Usage examples:

id('text').style.fontSize = '3em';
var x = id('text');
x.style.fontSize = '3em';
x.style.color = 'red';

function tags()

tags() is a small JavaScript function that manages collections of similar items. The function turns individual items on or off as required.

```
function tags(t, e) {
1. var x=0, y=document.getElementsByClassName(t), z=y.length;
2. for ( ; x < z; x++ ) {
3.     id(y[x].id).off(); }
4. id(e.id).on(); }
```

How does it work? A horizontal menu bar may have to be broken into menu fragments for different viewport sizes, as in the example below:

```
<nav>
<div id = 'm1'  class = 'menu'>etc.</div>
<div id = 'm2'  class = 'menu'>etc.</div>
<div id = 'm3'  class = 'menu'>etc.</div>
<div id = 'm4'  class = 'menu'>etc.</div>
<div id = 'm5'  class = 'menu'>etc.</div>
<div id = 'm6'  class = 'menu'>etc.</div>
</nav>
```

Only one menu fragment can be active at any one time. They all share the same class name (which doesn't have to appear in the CSS stylesheet), and they each have different ids. The best management strategy is to turn them all off, then turn on a nominated menu fragment.

Line 2 above sets up a loop that cycles through an array containing links to the six menu fragments.

Line 3 uses the *id()* function to turn off each menu fragment.

Line 4 uses the *id()* function to turn on the nominated menu fragment.

To turn on menu fragment number 4 the function would be called as follows:

```
tags('menu', m4)
```

The function is not part of the jRWD framework. It is included as a utility function to manage collections of items such as menus, sprites and carousels.

User-written functions

A JavaScript media query event-listener will fire when a breakpoint is reached, causing a JavaScript object to be sent to *event()* by the event-listener.

The object holds two important pieces of information: the text of the media query that triggered the event-listener; and a *true/false* value that indicates whether the viewport was getting bigger or smaller when the event occurred.

If the text of the JavaScript media query is 'min-width: 380px', *event()* will pass control immediately to *function px380*, a user-written function. *event()* will also pass the *true/false* value to *function px380* as an input parameter.

It follows then that for each breakpoint identified in the breakpoint array (line 3 in *setup()* above) there must be a user-written function in the web page's JavaScript code.

Each user-written JavaScript function must conform to the following rules:

1. Its name begins with the literal 'px' followed by a breakpoint number. The literal can be changed if the literal in line 12 of the framework is changed accordingly. For instance, if a framework user prefers function names that start with the literal 'pixel', then line 12 would look like this:
   eval ('pixel' + e.media.replace(/\D/g,'') + '(e.matches)');
2. It contains logic to reflect the viewport getting bigger at the breakpoint.
3. It contains logic to reflect the viewport getting smaller at the breakpoint.

For example, a user-written JavaScript function to switch the horizontal menu bar fragments at breakpoint 580 might look like this:

```
function px580(e) {
  if (e) tags('menu', m1);
  else tags('menu', m2);}
```

If flexbox is being used to control the web page's general layout and the CSS stylesheet contains a rule named **.flex** to change flexbox's presentation style from row mode to column mode at breakpoint 990, the user-written JavaScript function to change the browser layout would look like this:

```
function px990(e) {
  var x = css('.flex');
  if (e) x.style.flexDirection = 'row';
  else x.style.flexDirection = 'column'; }
```

The numerous CSS @media queries normally associated with responsive web design are replaced in jRWD by user-written JavaScript functions that supply the logic formerly vested in @media queries.

Generally speaking there will not be many user-written functions because there are few breakpoints in the breakpoint array at line 3 of the framework.

User-written JavaScript functions for jRWD are usually small and binary: do *this* if the viewport is getting bigger; do *that* if the viewport is getting smaller. This separation of logic from layout in jRWD makes web pages easier to build and maintain. It also results in a smaller overall code-base.

There is a distinction between CSS @media queries and user-written JavaScript functions that needs to be understood. User-written functions are binary: they put the web page either into state A or state B.

In sharp contrast, @media queries are layered on top of each other. @media query A can be superceded by @media query B, which can be superceded in turn by @media query C. If @media query C is removed @media query B automatically comes back into force without any intervention from the web developer. This is an important distinction between the way @media queries operate versus the binary operation of user-written JavaScript functions.

Organising and keeping track of multiple @media queries leads to complexity. Complexity is usually handled best by JavaScript.

Questions and answers

Q. How do I get jRWD?

A. jRWD is available at GitHub.

Q. How much does jRWD cost?

A. jRWD is open source software, published under the MIT licence.

Q. How do I contact the author?

A. Contact the author via jRWD's GitHub account.

Q. How do I install jRWD?

A. Download the two framework functions and the support functions from GitHub and place them in a .js file, say *jscript.js*. Put a link to *jscript.js* in the HTML file's <head> section. Put a link to the *setup function* in the HTML file's <body> tag. Those are the initial steps.

<script src = 'jscript.js'></script>
<body onload = 'setup()'>

Q. Are there additional steps?

A. Yes. Put breakpoint values in the breakpoint array at line 3 of the *setup function*, and write JavaScript event-handler functions that will process the breakpoints when they occur.

Q. What about customisation?

A. Read the comments in the *setup function* to establish how to customise the function. Also, make a decision about having *eval()* in or out of the working framework. Recommendation: leave it in.

Q. Do I have to do anything special in the CSS stylesheet?

A. No. The stylesheet should reflect a static view of the web page at a nominal viewport width. User-written JavaScript functions interact with the stylesheet to modify it as the viewport width changes.

Q. Will jRWD do my web page layouts for me?

A. Not directly, other than by manipulating the stylesheet. For page layout use CSS floats, or a conventional CSS grid layout, or better still, use the W3C's solution: flexbox.

Q. Does jRWD compete with Backbone, jQuery, Dodo, etc.?

A. No. jRWD is a lightweight, light-touch RWD solution. It provides facilities for manipulating the DOM and stylesheets. It does not do fade ins, fade outs, animation, or create forms or DOM objects.

Q. jQuery has a neat feature that allows me to inject HTML elements into the DOM on the fly. Can jRWD do this?

A. Not directly, but jRWD can simulate that behaviour. Put the elements into the HTML file and give each of them an id. jRWD can turn them on (*element.on()*) when they should be on view, and turn them off (*element.off()*) when they should be hidden.

Q. What about browser support?

A. jRWD is compatible with all modern browsers that have full JavaScript support. Internet Explorer 8-9 are not modern browsers.

Q. What about Opera Mini?

A. jRWD works on Opera Mini, but there are some caveats. Opera Mini does not have full JavaScript support. It uses remote servers to render a web page, and the servers send the rendition back to the mobile device to be displayed. The rendition may have minor flaws, but this is not caused by jRWD. It is standard Opera Mini behaviour.

Q. Is jRWD useful for Mobile First applications?

A. Yes, extremely so. Start with one breakpoint in the breakpoint array at line 3 in *function setup* and one user-written JavaScript function to process the breakpoint's event-listener. Progressively expand the design from there out to PC and laptop width.

Q. Can jRWD be applied to an existing, non-responsive website?

A. Yes indeed. First establish the breakpoints for the website, working from the widest view inwards. Progressively put each breakpoint into the breakpoint array and develop a user-written JavaScript function to process the event-listener. Some HTML elements may have to be given ids so they can be turned on and off, and some new CSS rules may be needed in the stylesheet to cater for the changed circumstances.

Q. Is there an easy way to establish the breakpoints?

A. Yes. Modify the non-responsive web page as follows:
put **onresize = 'resize()'** in the <body> tag;
put ** ** after the <body> tag;
put **<script>**
function resize(){size.textContent=window.innerWidth}
</script>
in the <head> section, straight after the existing <script> tag. The browser will now display its width when it is resized.

Q. I don't feel comfortable using *eval()*. Can I strip it from jRWD?

A. Yes. There are instructions for replacing *eval()* here.

Q. Can I see a working jRWD website?

A. Yes, here. Be sure to view the website on a PC and on a mobile device. Then look at the source HTML, the CSS stylesheet, and the JavaScript code. If the mobile device is running iOS9 the website may not render properly. This is caused by iOS9 dropping support for flexbox, which the website uses to control column layout.

Q. Is there a way to prevent responsive web pages from jumping around when the viewport is being stepped down on mobile devices?

A. Yes, there is. Follow these steps:
put **class = 'hide'** in the <body> tag;
put **.hide {margin : -5000px;}** in the stylesheet;
and put **css('.hide').style.margin = 0;** after line 11 in the *setup function*. The stepping-down process will now occur off-screen.

How evil is *eval()*?

In his 2008 book ***JavaScript: The Good Parts***, author Douglas Crockford warned against using JavaScript's *eval()* function:

> *The eval function and its relatives (Function, setTimeout, and setInterval) provide access to the JavaScript compiler. This is sometimes useful, but in most cases it indicates the presence of extremely bad coding. The eval function is the most misused feature of JavaScript.*

In the ensuing years Crockford's injunction has been the subject of intensive debate. Some agree with him, others don't. *eval()* has been part of JavaScript since its inception. Some frameworks use it, as does a JSON parser written by a person named Douglas Crockford (json2.js, has an *eval()* at line 504).

John Resig, author of the original version of jQuery, devoted chapter 9 of his 2013 book ***Secrets of the JavaScript Ninja*** to run-time code evaluation, which was mainly a discussion of *eval()* and *Function*. His concluding advice:

> *While the potential for misuse of this powerful feature is possible, the incredible power that comes with harnessing code evaluation makes it an excellent tool to wield in our quest for JavaScript ninja-hood.*

Ironically, one of the ways to avoid using *eval()* is to use *Function*, as will be revealed below. *Function* is also on Douglas Crockford's banned list

A blanket ban on using *eval()* is clearly absurd; instead the focus should be on: when is it safe/unsafe to use *eval()*?

The general consensus is that *eval()* is unsafe when used on back-end servers and when used with unknown, user-supplied data. When used in browsers with browser-supplied data it is as safe as any other JavaScript function.

jRWD uses *eval()* in two places: in line 11 of *setup()* where it is used with known parameters; and in *event()* (line 12) where it is used with data supplied by a call-back function. Both instances are perfectly safe and necessary.

However, *eval()* can be removed from jRWD easily if required.

Follow these three steps:

1. Change **eval**(....) in line 11 to **val**(....).

2. Discard line 12 from *event()* and replace it with:
   val('px'+e.media.replace(/\D/g,'')
   +'('+(e.matches? 'true' : 'false')+')');

3. Add this one-line function to the framework:
   function val (e) { return new Function (e) () }

Appendix A

This section is a graphical analysis of data found at this website. The data are presented here as a ready-reference for responsive web design.

The boxes show various mobile device configurations arranged by viewport widths. For smartphones, portrait widths are more standardised than are landscape widths; for tablets the reverse is true.

Tablets

P	L
600	**960**

P	L
768	**1024**
600	
496	

P	L
800	**1280**
768	
720	
648	

Smartphones

P	L
	427
	400
240	**320**

P	L
	569
	568
	545
	544
	534
	533
	505
	496
320	**480**
	427
	415
	406
	401

P	L
360	**640**
	598
	559
	480

P	L
384	**640**
375	**667**
412	**732**
414	**736**

Appendix B

Mobile devices		
Apple iPad	768	1024
Apple iPhone 3/4	320	480
Apple iPhone 5	320	568
Apple iPhone 6	375	667
Apple iPhone 6 Plus	414	736
BlackBerry PlayBook	600	1024
BlackBerry Z30	360	640
Google Nexus 10	800	1280
Google Nexus 4	384	640
Google Nexus 5	360	640
Google Nexus 6	412	732
Google Nexus 7	600	960
LG G3/G4/Nexus 5	360	640
LG Optimus L70	384	640
Nokia Lumia 520	320	533
Nokia N9	360	640
HTC One	360	640
HTC 8X	320	480
Microsoft Lumia	320	480
Samsung Galaxy Note	400	640
Samsung Galaxy S2	320	533
Samsung Galaxy Nexus	360	600
Samsung Galaxy S3/S4/S5	360	640
Sony Xperia Z3	360	598
Sony Xperia Z	360	640

The tablets and smartphones in the table above were selected to represent the spectrum of mobile devices currently available.