INSTANT

Short | Fast | Focused

Zepto.js

Create fast and responsive mobile web apps with Zepto.js

Ian Pointer

[PACKT] PUBLISHING

Instant Zepto.js

Create fast and responsive mobile web apps with Zepto.js

Ian Pointer

[PACKT]
PUBLISHING

BIRMINGHAM - MUMBAI

Instant Zepto.js

First published: September 2013

Production Reference: 1230913

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-289-0

www.packtpub.com

Credits

Author

Ian Pointer

Reviewer

Ezekiel Chentnik

Acquisition Editor

Antony Lowe

Commissioning Editor

Sharvari Tawde

Technical Editors

Pankaj Kadam

Gaurav Thingalaya

Project Coordinator

Amigya Khurana

Proofreader

Stephen Copestake

Production Coordinator

Nitesh Thakur

Cover Work

Nitesh Thakur

Cover Image

Ronak Dhruv

About the Author

Ian Pointer is a Senior Developer at Open Software Integrators in Durham, NC, USA. Originally from the United Kingdom, he has been living in America for two years and has been doing full-stack development on high-traffic websites on both sides of the country.

I'd like to thank my partner, Stacie, for her patience when I was writing this while we were moving house, my friend Tammy for all her valuable input and encouragement, and my family back home in Britain for words of wisdom along the way.

About the Reviewer

Ezekiel Chentnik is that awesome developer who can do both developing and designing well. In a sea of developers and designers very few have the ability to excel on all levels. You are either a designer or a developer but rarely both. He has a knack for elegant code—reusable, responsive, and user-friendly designs—and knows how to handle enterprise level projects. He has 8 plus years' experience in front-end development, back-end development, design, responsive web design, mobile app development and design, and raw JavaScript development. He is a JavaScript whiz-kid and whatever the challenge is, he takes it. He is passionate about JavaScript development, and is constantly pushing the limit. His recent projects include some of his favorite JavaScript libraries: Zepto.js, Backbone.js, Underscore.js, Marionette.js, and Modernizr.js.

www.packtpub.com

Support files, eBooks, discount offers and more

You might want to visit www.packtpub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

Table of Contents

Instant Zepto.js

Welcome to *Instant Zepto.js*. This book has been especially created to provide you with all the information that you need to get accustomed to Zepto.js. You will learn the basics of Zepto.js, get started with building your first course, and discover some tips and tricks for using Zepto.js.

This document contains the following sections:

So, what is Zepto.js? finds out what Zepto.js actually is, what you can do with it, and why it's so great.

Installation teaches you how to download and install Zepto.js with minimal fuss and then set it up so that you can use it as soon as possible.

Quick start – the Zepto.js API introduces you to the main features of Zepto.js and how to use them in your web pages.

Top 3 features you need to know about covers how to perform three tasks with the most important features of Zepto.js. By the end of this section, you will learn how to use Zepto.js for animating web pages, building a custom version of the library, implementing touch features on a mobile device, and extending Zepto.js with plugins.

People and places you should get to know involves open source projects that are centered around a community. This section provides you with many useful links to the project page and forums as well as a number of helpful articles, tutorials, blogs, and the Twitter feeds of Zepto.js super-contributors.

Also, you'll find all the code samples in this book at `https://github.com/falloutdurham/zeptojstarter`.

So, what is Zepto.js?

One of the most influential JavaScript libraries in the last decade of web development is jQuery, a comprehensive set of functions that make **Document Object Model** (**DOM**) selection and manipulation consistent across a range of browsers, freeing web developers from having to handle all these themselves, as well as providing a friendlier interface to the DOM itself.

Zepto.js is self-described as an aerogel framework—a JavaScript library that attempts to offer the most of the features as the jQuery API, yet only taking up a fraction of the size (9k versus 93k in the default, compressed current versions Zepto.js v1.01 and jQuery v1.10 respectively). In addition, Zepto.js has a modular assembly, so you can make it even smaller if you don't need the functionality of extra modules. Even the new, streamlined jQuery 2.0 weighs in at a heavyweight 84k.

But why does this matter?

At a first glance, the difference between the two libraries seems slight, especially in today's world where large files are normally described in terms of gigabytes and terabytes. Well, there are two good reasons why you'd prefer a smaller file size. Firstly, even the newest mobile devices on the market today have slower connections than you'll find on most desktop machines. Also, due to the constrained memory requirements on smartphones, mobile phone browsers tend to have limited caching compared to their bigger desktop cousins, so a smaller helper library means more chance of keeping your actual JavaScript code in the cache and thus preventing your app from slowing down on the device. Secondly, a smaller library helps in response time—although 90k versus 8k doesn't sound like a huge difference, it means fewer network packets; as your application code that relies on the library can't execute until the library's code is loaded, using the smaller library can shave off precious milliseconds in that ever-so-important time to first-page-load time, and will make your web page or application seem more responsive to users.

Having said all that, there are a few downsides on using Zepto.js that you should be aware about before deciding to plump for it instead of jQuery. Most importantly, Zepto.js currently makes no attempt to support Internet Explorer. Its origins as a library to replace jQuery on mobile phones meant that it mainly targeted WebKit browsers, primarily iOS. As the library has got more mature, it has expanded to cover Firefox, but general IE support is unlikely to happen (at the time of writing, there is a patch waiting to go into the main trunk that would enable support for IE10 and up, but anything lower than Version 10 is probably never going to be supported). In this guide we'll show you how to include jQuery as a fallback in case a user is running on an older, unsupported browser if you do decide to use Zepto.js on browsers that it supports and want to maintain some compatibility with Internet Explorer.

The other pitfall that you need to be aware of is that Zepto.js only claims to be a jQuery-like library, not a 100 percent compatible version. In the majority of web application development, this won't be an issue, but when it comes to integrating plugins and operating at the margins of the libraries, there will be some differences that you will need to know to prevent possible errors and confusions, and we'll be showing you some of them later in this guide.

In terms of performance, Zepto.js is a little slower than jQuery, though this varies by browser (take a look at `http://jsperf.com/zepto-vs-jquery-2013/` to see the latest benchmark results). In general, it can be up to twice as slow for repeated operations such as finding elements by class name or ID. However, on mobile devices, this is still around 50,000 operations per second. If you really require high-performance from your mobile site, then you need to examine whether you can use raw JavaScript instead—the JavaScript function `getElementsByClassName()` is almost one hundred times faster than Zepto.js and jQuery in the preceding benchmark.

Installation

Installing Zepto.js is very straightforward. The following are the steps that will help you install Zepto.js:

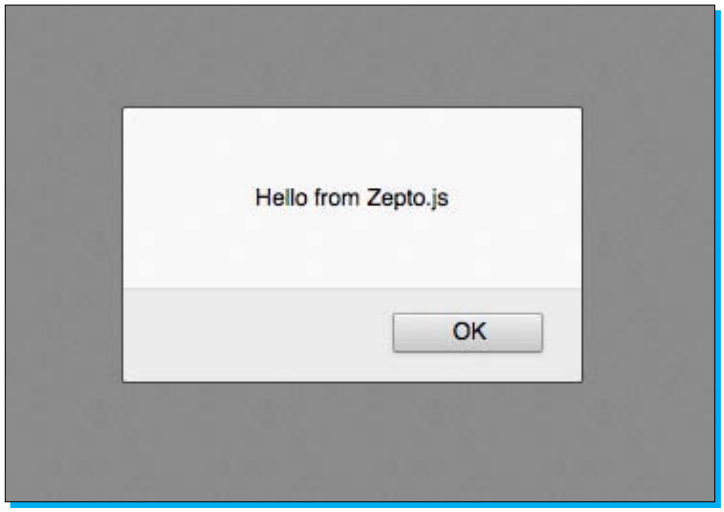
1. The latest minified default build (which includes the most of what you'll need for day-to-day web development, though later on, you'll learn how to build a custom version of the library with extra or fewer modules) can always be found at `http://zeptojs.com/zepto.min.js`.
2. If you need it, the uncompressed JavaScript build can be found at `http://zeptojs.com/zepto.js`.
3. Or you can clone the Git repository at GitHub by issuing this Git command on your command line `git clone https://github.com/madrobby/zepto.git`.
4. Once you have obtained the `zepto.js` file (we'll be working with the `zepto.min.js` file in our examples), all you need to do to enable it is include the following piece of code in your HTML pages in the normal way:

```
<html>
  <body>
    <script src="/path/to/zepto.min.js"></script>
  </body>
</html>
```

5. Or, to let you know that it really is there:

```
<html>
  <body>
    <script src="/path/to/zepto.min.js"></script>
    <script>$(function ($)
    { alert("Hello from Zepto.js")})</script>
  </body>
</html>
```

6. Open this page in either Firefox, Chrome, or Safari, and you should see something like the following screenshot:



Having installed Zepto.js, the next section will detail the ways that it can be used to select and manipulate DOM objects, CSS rules, set up events, and send Ajax requests.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Quick start – the Zepto.js API

Just as in jQuery, use of Zepto.js revolves around the `$` object. In your web applications, you will build up collections using the `$()` functions, and having built these collections up, you will do things like alter their HTML content, add and remove CSS classes, set up and tear down event handlers, and make Ajax requests. The most basic operation is creating a collection of DOM nodes by passing a CSS selector to `$()`, for example, `$('#my-id')`. This will capture the DOM element on the page with the ID of `my-id`. The following are the some of the examples:

- The `$("div")` function will select all `<DIV>` elements on the page
- The `$("div.my-class")` function will select all the `<DIV>` elements with a class of `my-class`
- The `$("div.my-class p")` function will select all the `<P>` elements within `<DIV>` elements that have a class of my-class
- The `$("div.my-class p:first")` function will select the first `<P>` elements within all the `DIV` elements of `my-class`

If you need a refreshing on CSS selectors, head over to the W3C's page at `http://www.w3.org/TR/css3-selectors`. The `$()` object can also be used to create new DOM nodes: `$("<p>Hi</p>")` or `$("<p />", {text: "Hi"})`. Finally, you can also pass a function to `$()`, and it will be called when the `DOMContentLoaded` event is fired by the browser rendering the HTML page:

```
$(function($) {
  alert("page loaded!");
})
```

Step 1 – DOM filtering

Zepto.js provides a series of filter methods that can be used to refine collections of DOM nodes using the concept of chainable methods. Here are a few examples.

```
add(selector, [context])
```

This modifies the current collection by applying the CSS selector to the entire page, or within the context supplied by the optional `context` parameter. For example, to add all `<DIV>` elements within a class of `"form-class"` to an existing group of form elements, you would use the following code:

```
$("form").add("div", ".form-class")
```

And if you wanted to add all the `<DIV>` elements on the page, you would omit the final `context` parameter `$("form").add("div")`.

```
first()
last()
next([selector]
prev([selector])
```

The `first()` and `last()` functions return the first and last elements of the collection, so `$('button').last()` would provide the final button on the page. The `next()` and `prev()` functions will return the next and previous elements in the collection, but can also be supplied a CSS selector that returns the next or previous element in the collection that matches the selector.

```
parent([selector])
parents([selector])
siblings([selector])
children([selector])
```

These similar functions allow you to discover the parents, children, and siblings of a collection with ease. The `parent()` function will find the immediate parent of a collection, with a selector, the immediate parent that matches the selection criteria. However, the `parents()` function will traverse the DOM tree all the way up to the `<HTML>` element, filtered by the optional `selector` parameter. The `children()` and `siblings()` functions will provide the children and siblings of the collection, again filtered by the optional `selector` criteria. Finding all the children of an element with children that match a class of `required-class` would be `$("#target-id").children('.required-class')`.

```
find(selector)
find(collection)
find(element)
```

The `find()` function takes either a selector, a collection, or a DOM element, and will return a match found in the scope of its parameter. For example, `$("table").find("br")` would return all the `<BR>` tags in all the `<TABLE>` elements on the page.

```
filter(selector)
filter(function(index){ ... })
not(selector)
not(collection)
not(function(index){ ... })
```

The `filter()` function will return a Zepto.js collection which only contains the items that match the CSS selector supplied as a parameter. Alternatively, you can supply a function as an argument, and `filter()` will only include an element in the collection it returns when that function returns a `true` value. The `not()` function does the converse, returning a collection that does not contain the selector or collection, or when the optional function supplied returns `false`.

Step 2 – DOM manipulation

Having obtained a collection of DOM nodes, you'll often want to perform some modification on them such as setting some text or altering some CSS properties. Zepto.js provides functions for this that are chainable in the same way as the filtering functions, which means you can build a complex set of queries and changes in just a single line of JavaScript. For example, perhaps you need to change all the `<DIV>` elements on a page from `my-class` to `that-class`. With Zepto.js, this would be done using the following piece of code:

```
$(".my-class").each(function() { this.removeClass;
this.addClass("that-class")}
```

The `each()` function supplies two optional parameters to the parameter function—`index` and `item`, to provide more fine-grain control if required. In addition, if the function inside `each()` returns `false`, the iteration stops.

```
html()
html(content)
html(function(index, oldHtml){ ... })
```

The `html()` function will return the HTML of the elements in the collection if it is called without a parameter, but when supplied in a string, that string is used to set the `innerHTML` property of each element in the collection. When a function is given as a parameter, that function is called on each element, with `index` and `oldHtml` being given as parameters for use in the function. A simple example would be `$("#replace").html("<p>new HTML content</p>").`

```
text()
text(content)
```

Similar to the `html()` function, the `text()` function will either get or set the `textContent` property on every element in the collection it is being applied to.

```
after(content)
before(content)
prepend(content)
```

The first two functions will add content either after or before elements in the collection. To add a `<BR>` element before every `<FORM>` element, you would use `$("form").before("<br>").`

The content supplied to the `before()` and `after()` functions can be a string as in the preceding sentence, DOM nodes, or an array of nodes. The `prepend()` function will prepend the given content to the DOM in each element of the collection.

```
prop(name)
prop(name, value)
prop(name, function(index, oldValue){ ... })
```

The `prop()` function will get or set an attribute on DOM elements in a collection. You can also provide a function instead of a value to iterate over the collection.

```
wrap(structure)
wrap(function(index){ ... })
wrapAll(structure)
wrapInner(structure)
wrapInner(function(index){ ... })
```

The wrap functions allow you to take a Zepto.js collection and provide wrapping DOM elements for either each element in the collection, or the collection itself. For example, to wrap every `<P>` element in a list element, you would write `$("p").wrap("<li></li>")`.

To wrap all `<li>` elements on the page into an ordered list, you would use the `wrapAll()` function and the code for this would be `$("li").wrapAll("<ol></ol>")`.

In addition, the `wrapInner()` function works in a similar manner, except it wraps the `contents` of each item separately in the collection, rather than the item itself.

Both `wrap()` and `wrapInner()` can also take a function as a parameter, allowing you to have more control over the wrapping process if desired.

Step 3 – CSS operations

Zepto.js also provides a selection of functions for operating on CSS attributes and classes, including a few helper methods for positioning and altering classes.

```
addClass(name)
addClass(function(index, oldClassName){ ... })
hasClass
removeClass([name])
removeClass(function(index, oldClassName){ ... })
```

This group of functions operates on the CSS class names on the supplied collection, either adding a class to an element, determining whether an element has a specified class, or removing a class.

If `remoteClass` is called without an argument, it will remove all the classes on all elements in the collection.

```
show()
hide()
toggle()
```

These helper functions will alter the CSS display attribute. The `hide()` function will set all the elements in the collection to `display: none`, and `show()` will restore all the element's `display` setting. The `toggle()` function flips between the two states, so you don't have to handle both cases yourself.

```
height()
height(value)
height(function(index, oldHeight){ ... })
width()
width(value)
width(function(index, oldWidth){ ... })
```

The `height()` and `width()` functions will either return the height and width CSS properties of the elements in the collection or, if given a value or a function, it will set the height and width properties to either the value or what the function returns.

```
css(property)
css(property, value)
css({ property: value, property2: value2, ... })
```

However, when full control over a collection's CSS is required, `css()` provides access to the collection's CSS. Any CSS property can be queried by supplying it, and any property can be set by supplying a value. If the `css()` function is provided with a hash of key/value pairs of property/setting, it will set all the properties in one fell swoop. Setting the font and color CSS rules on all list elements with a class of `incomplete` could be handled by `$("li.incomplete").css({ 'font-family': 'Gotham', 'color': 'red'})`.

Step 4 – DOM event handling

A possible source of confusion when using Zepto.js is that there appears to be a myriad of different ways to handle events on DOM objects. This is mainly due to the library's intent of providing a similar interface to jQuery, which has gone through several different recommended methods of binding event handlers over the years (for example, `bind()`, `live()`, and more). jQuery has recently deprecated many of these functions, so while knowing that they exist is useful, they should no longer be used. Instead, event handling should be carried out only using the `on()` and `off()` functions.

```
on(type, [selector], function(e){ ... })
on({ type: handler, type2: handler2, ... }, [selector])
```

The `on()` function will set up event handling for all the events specified in the `type` string (with events being separated by spaces, or via a set of key/value strings in a hash). If a CSS selector is passed in as an additional parameter, the event handler will only fire if the element in question matches that selector. For example, `$("#typical-event").on('click', function(e) { alert("event fired!"}, ".event-creator")` will attach a `click` event handler to the `#typical-event` DOM object, but will only fire if the click happens on a child element with the class `event-creator`.

```
off(type, [selector], function(e){ ... })
off({ type: handler, type2: handler2, ... }, [selector])
off(type, [selector])
off()
```

This function is the counterpart to `on()`. Calling `off()` will remove every event handler in the collection. Otherwise, you can provide the types of handlers to remove in either a space-delimited string or a hash of event types as before. The optional `selector` will limit the removal of event handlers to only those items in the collection that match the selector; if you provide a function, only that function will be removed from the event handler (the default operation is to remove all functions associated with an event type if a function is not supplied).

Switching off click event handling for all list elements in a list that have a class name of `deactivated` would be done by `$("#list-of-things").off('click', 'li.deactivated')`.

```
one(type, function(e){ ... })
one({ type: handler, type2: handler2, ... })
```

Sometimes you only want an event to fire once and then never have to deal with it ever again; Zepto.js provides a way of doing this with `one()`. Taking the same parameters as `on()`, the library will remove the handler after it has been triggered without you having to explicitly make any calls to `off()`.

```
trigger(event, [data])
```

If an event needs to be triggered specifically by your application, you can use the `trigger()` function on a collection. This will trigger events specified in the space-delimited string of `type` to all elements of the collection, with an optional `data` parameter that, if present, will be passed to the handling function that is triggered.

Step 5 – Ajax requests

One of the major uses of jQuery is for its `$.ajax` function, which papers over the differences in the implementation of `XMLHttpRequest()` across browsers and makes asynchronous HTTP requests a simple affair. Zepto.js provides a similar API, though for those of you who use jQuery, Zepto.js does not use the promises facility present in newer versions of the jQuery library. This means that callbacks such as `done()`, `fail()`, and so on will not work in Zepto.js.

```
$.ajax(options)
```

The `$.ajax` function is deceptively simple, as the `options` hash contains all the details needed for everything from the type of Ajax request being sent to callbacks that are set up for success and errors. The options, taken from the Zepto.js documentation, are:

- `type (default: "GET")`: This specifies the type of HTTP request method (`GET`, `POST`, or other)
- `url (default: current URL)`: This specifies the URL to which the request is made
- `data (default: none)`: This specifies data for the request; for `GET` requests it is appended to query string of the URL
- `processData (default: true)`: This specifies whether to automatically serialize data for requests not from `GET` to string
- `contentType (default: "application/x-www-form-urlencoded")`: This specifies the content type of the data being posted to the server (this can also be set via headers). Pass `false` to skip setting the default value
- `dataType (default: none)`: This specifies the response type to expect from the server (`json`, `jsonp`, `xml`, `html`, or `text`)
- `timeout (default: 0)`: This specifies the request timeout in milliseconds, `0` for no timeout
- `headers`: This specifies the object of additional HTTP headers for the Ajax request
- `async (default: true)`: This can be set to `false` to issue a synchronous (blocking) request
- `global (default: true)`: This specifies triggering global Ajax events on this request
- `context (default: window)`: This specifies the context to execute callbacks in

In addition, functions can be supplied in the option's hash as callbacks for various stages of the `XMLHttpRequest`. Again, from the documentation:

- `beforeSend(xhr, settings)`: This callback function is used before the request is sent. It Provides access to the `xhr` object. Return `false` from the function to cancel the request
- `success(data, status, xhr)`: This is the function to be called when request succeeds
- `error(xhr, errorType, error)`: This is the function to be called if there is an error (timeout, parse error, or status code not in HTTP 2xx)
- `complete(xhr, status)`: This is the function to be called after the request is complete, regardless of error or success

Thankfully, you won't need to fill out all the options for every request that you send. For example, sending a `POST` request with the `JSONP` data with `$.ajax` can be performed in just a few lines:

```
$.ajax({
  type: 'POST',
   url: '/update_users',
   data: JSON.stringify({ user: 'John Smith', id: 4242 }),
   contentType: 'application/json'
})
```

Of course, you will probably want to do something with what the server responds with, so a more useful call would look something more like the following piece of code:

```
$.ajax({
  type: 'POST',
  url: '/update_users',
  data: JSON.stringify({ user: 'John Smith', id: 4242 }),
  contentType: 'application/json',
  success: function(data) {
    console.log(data)
  }
  error: function(xhr, type){
    alert('Ajax error!')
  }
})
```

This will send the response from the server to the browser's debug console on success, and generate an alert if something goes wrong in the request (for example, if the server is not responding).

Top 3 features you need to know about

Animation

One of the most common uses of jQuery has traditionally been for animation on web pages and applications. Functions such as `fadeTo()`, `slideIn()`, and `slideOut()` have formed the backbone of many web applications over the past decade. But, time has moved on somewhat and the W3C has incorporated animation and transitions into the CSS3 specification, so the need for the custom JavaScript helpers that jQuery provides has lessened.

Because of the new features present in CSS3, Zepto.js does not include these methods by default (though, once you learn how to build a custom build later in this section, you can include the `fx_methods` module to reinclude them). Instead, the `animate()` function of Zepto.js provides easy access to the underlying CSS animations and transitions that are present in almost all browsers available today.

CSS animations and transitions can operate on a large set of CSS properties—a good resource that demonstrates what you can do with these properties can be found on Lea Verou's page at `http://leaverou.github.io/animatable`. The animation system of Zepto.js is similar in use to other operations, working on a collection of Zepto.js objects:

```
animate(properties, [duration, [easing, [function(){ ... }]]])
animate(properties, { duration: msec, easing: type, complete: fn
})
animate(animationName, { ... })
```

The first argument in all three usages is what is going to be animated—either a hash of CSS rules, or a CSS animation keyframe name that has been specified in your CSS rules elsewhere. You can supply a duration in milliseconds, or `fast (200ms)`, `slow (600ms)`, or even define your own speeds by adding properties to `$.fx.speeds['new-speed-name'] = value`. Otherwise, the default speed for animations is 400 ms (which can also be changed if you redefine `$.fx.speeds['_default']`. The second form of the function is probably better to use than the first, as it means you don't have to keep track of which parenthesis goes where when you come back to look at your code at a future point.

The easing parameter allows you to specify a choice of easing functions for the tweening of your animation. These are the standard ones that are defined in CSS: `ease`, `ease-in`, `ease-out`, `linear`, `ease-in-out`, plus a `cubic-bezier` curve so you can define your own timing curve (have a look at `http://matthewlein.com/ceaser` to play around with generating different timing curves). Finally, the last parameter, or the `complete` property, allows you to define a function which will be called when the animation ends.

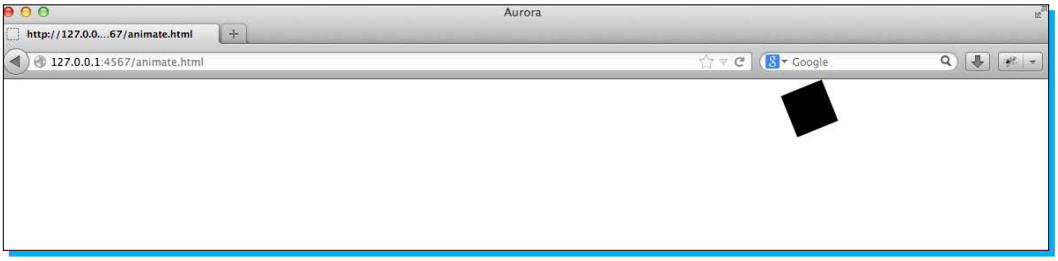
Here's a simple page that shows off the animation capabilities (you can also find it online at `https://github.com/falloutdurham/zeptojstarter`). When the page loads, it will shoot off to the right side of the screen, and log `finished` to the browser console.

```
<html>
  <body>
  <style>
    .box {
      width: 50px;
      height: 50px;
      background-color: black;
      left: 0px;
      position: absolute;
    }
  </style>
    <div class="box"></div>
    <script src="zepto.js"></script>
    <script>
    Zepto(function($){
      $(".box").animate(
        {'left': '100%'},
        { easing: 'ease-in',
        duration: 1000,
        complete: function(){ console.log("finished!")}
        })
      });
    </script>
  </body>
</html>
```

And of course, we can animate multiple properties at once. In the preceding example, if you change the animate call to look like the following piece of code:

```
$(".box").animate(
  { 'left': '100%',
  'rotateZ': '360deg'},
  { easing: 'ease-in',
   duration: 1000,
  complete: function(){ console.log("finished!")}
});
```

Then, when the page is reloaded, it will spin a full 360 degrees as it heads towards the right-hand side of the screen as shown in the next screenshot:



There'll be some examples later on in this section that combine animation with events such as tapping and swiping on the screen.

The build system – building a custom Zepto.js

Why would you want to build a custom version of Zepto.js? Firstly, to keep the size of the library as small as possible, the default version of Zepto.js that comes with the Git repository or the one linked on the Zepto.js website doesn't include all the possible modules that the library can use. In particular, touch and gesture support aren't included by default. Secondly, if, for example, you're not using forms on your web app, then loading the default library will include code that you're not going to use, taking up precious milliseconds of loading and parsing time. But, thanks to the custom build system, you don't have to include parts that you're not going to use.

In this section, we are going to build a custom version of Zepto.js that includes touch and gesture support, which we'll use in the next section to show how to use the library to create a touch interface on mobile devices.

So just follow these simple steps:

1. First, you'll need to have Node.js installed on your machine to build a custom library. The Node.js website has binary and source packages for most systems and they can be found at `http://nodejs.org/download`.
2. Once that's installed, go into the `Zepto.js` directory and type the following command:

   ```
   $> npm install
   ```

3. That will download all the Node.js requirements that are needed to build (and test) Zepto.js.
4. To build a default version of Zepto.js, you would type:

   ```
   $> npm run-script dist
   ```

5. That will give you output on the terminal that should look something like the following command-line output:

```
$> zepto@1.0.0 dist /Users/ianpointer/tmp/zepto

$> coffee make dist

dist/zepto.js: 54.6 KiB

dist/zepto.min.js: 26.6 KiB

dist/zepto.min.gz: 9.7 KiB

compression factor: 5.6
```

6. This allows you to see how much your build of Zepto.js minifies, and how much it would compress further if your server is equipped to handle gzip compression.
7. If you open up `dist/zepto.js` in your favorite text editor, the comment on the first line should tell you what modules have been installed:

```
/* Zepto v1.0-7-g579f376 - polyfill zepto detect event ajax
form fx - zeptojs.com/license */
```

8. This is the default selection of modules. But we're going to remove the `ajax` and `form` modules, replacing them with `touch` and `gesture` (we could, of course, just include all the modules listed in the `README` file if we so desired).
9. To do this, we set the `MODULES` environment variable to list what modules we want to be included:

```
$> export MODULES="polyfill zepto detect event fx touch gesture"

$> npm run-script dist

$>zepto@1.0.0 dist /Users/ianpointer/tmp/zepto

$> coffee make dist

dist/zepto.js: 47.4 KiB

dist/zepto.min.js: 22.1 KiB

dist/zepto.min.gz: 8.1 KiB

compression factor: 5.9

$> head -1 dist/zepto.js

/* Zepto v1.0-7-g579f376 - polyfill zepto detect event fx
touch gesture - zeptojs.com/license */
```

We now have a custom version of Zepto.js that includes touch and gesture support. Rename this to `zepto-touch.js` for the next section.

Mobile device support

As the origins of Zepto.js lie in producing a slimmer jQuery-like interface for web applications on mobile devices, you would expect that the library comes with a host of features aimed at them. However, as Zepto.js has moved towards being a more general library, some of these features are not enabled by default and have to be added by producing a custom build as we saw in the last section.

Device and browser detection

The `detect` module (included in the default distribution) provides a suite of boolean variables that your web app can test for to gather information about the browser/device it is running on. Here's what Zepto.js gives you:

```
$.os.phone
$.os.tablet
$.os.ios
$.os.android
$.os.webos
$.os.blackberry
$.os.bb10
$.os.rimtabletos
$.os.iphone
$.os.ipad
$.os.touchpad
$.os.kindle
$.browser.chrome
$.browser.firefox
$.browser.silk
$.browser.playbook
```

As you can see, it allows you to drill down fairly deeply into the environment your application is running on. One slight wrinkle is that these variables can be undefined instead of false so, when using them, you'll want to preface them with `!!` to make sure you get a proper boolean result. For example, `!!$os.ios` will definitely return false on all Android devices.

Touches and gestures

The `touch` and `gesture` modules allow Zepto.js to respond to touch events. The `touch` module provides tapping and swiping support, while the `gesture` module provides access to pinching events, but only on iOS devices.

Here's a web page with some basic CSS rules to create three circles, as well as defining the viewport so the browser doesn't try to resize the page to fit its screen. We'll add some code to show off the touch events.

```
<!doctype html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,user-
scalable=no">
  </head>
  <body>
    <style>
      circle {
        position: relative;
        margin: auto;
        width: 3em;
        height: 3em;
        border-radius: 50%;
        margin: 10%;
      }
      #one {
        background-color: red;
      }
      #two {
        background-color: green;
      }
      #three {
        background-color: blue;
      }
      .p {
        position: relative;
        color: white;
        margin-top: 4em;
      }
    </style>
    <div id="one" class="circle"><p></p></div>
    <div id="two" class="circle"></div>
    <div id="three" class="circle"></div>
    <script src="zepto-touch.js"/></script>
  </body>
</html>
```

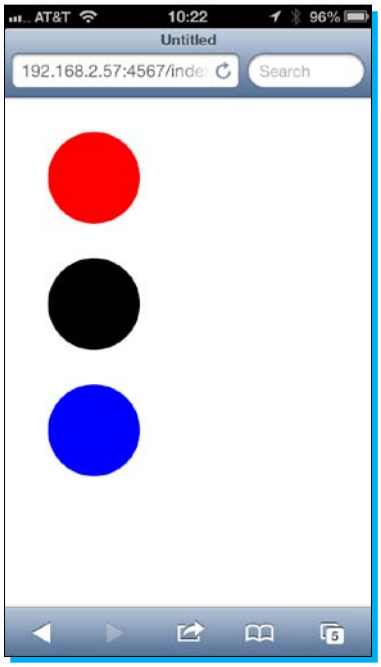
Firstly, we'll add some code to change the color of a circle to black when a tap event is fired on touching a circle. We can do that in two ways, either passing the event to a normal `on()` function call, or using one of Zepto.js's convenience functions which call `on()` behind the scenes (`tap()`, `doubleTap()`, `singleTap()`, `longTap()`, `swipe()`, `swipeLeft()`, `swipeRight()`, `swipeUp()`, and `swipeDown()`). So below the script tag that loads Zepto.js, we could either write:

```
<script>Zepto(function($){  $(".circle").tap(function () {
$(this).css('background-color', 'black');})   })</script>
```

or

```
<script>Zepto(function($){  $(".circle").on('tap',function () {
$(this).css('background-color', 'black');})   })</script>
```

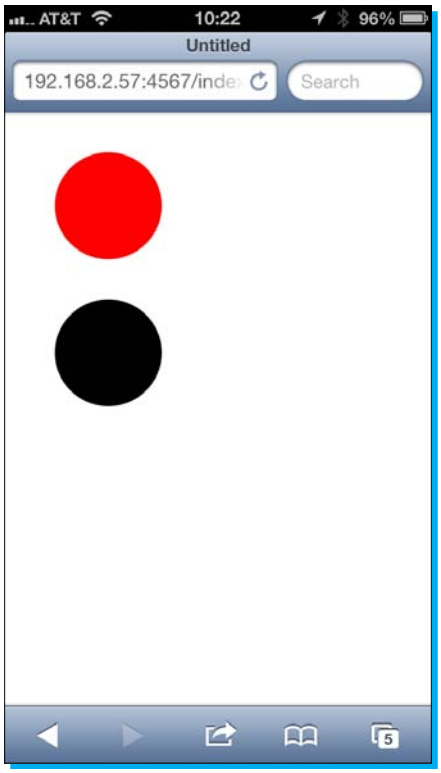
The following screenshot shows how it will appear:



If we wanted to pop the circle on a double-tap, we would change the script tag to be:

```
<script>Zepto(function($){
  $(".circle").on('singleTap',function () {
  $(this).css('background-color', 'black');});
  $(".circle").on('doubleTap',function () { $(this).hide() ; });
})
</script>
```

You have to handle both the `singleTap` and `doubleTap` events separately if you want to handle both single and double-tap events; if you just used `tap()`, it would fire on both tap events of a user double-tapping on an element on your page, which is probably not what you intended. Have a look at the following screenshot:



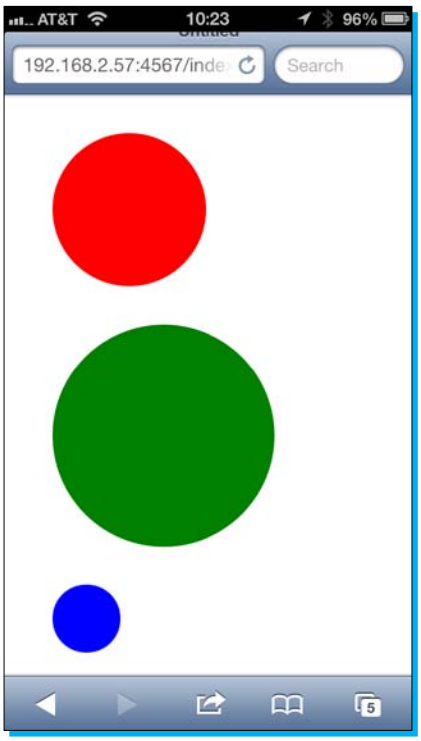
Gestures

The `gesture` module adds iOS's pinching gestures in the form of `pinch`, `pinchIn`, `pinchOut` events. We'll use the animation features of Zepto.js to make the circles bigger when a user makes an opening pinch gesture, and make them smaller when the closing pinch gesture is recognized. Thanks to the animation features, this should create a smooth transition as the circles grow or shrink.

```
<script>Zepto(function($){
  if (!!$.os.ios) {
    $(".circle").on('pinchOut', function(){
      new_width_and_height = $(this).width() * 1.5;
    $(this).animate({
```

```
        width: new_width_and_height,
        height: new_width_and_height
      });
      });
      $(".circle").on('pinchIn', function() {
        new_width_and_height = $(this).width() / 1.5;
        new_border_radius = new_width_and_height / 2;
      $(this).animate({
        width: new_width_and_height,
        height: new_width_and_height
      });
      });
    }});
  </script>
```

The following screenshot shows how it will appear:



Of course, as well as just making circles move about a screen, the gesture support can be used to build up a fairly complex user interaction. In this section, we'll build a basic table view that looks similar to an iPhone's native view. Firstly, you'll need an unordered list, and some list item elements inside that list:

```
<ul>
  <li class="list-element">One</li>
  <li class="list-element">Two</li>
  <li class="list-element">Three</li>
</ul>
```

Then, some CSS rules to make it look less like a traditional list and more like the TableView interface:

```
ul {
  width: 100%;
  margin: 0 0 0 0;
  padding: 0 0 0 0;
}
.list-element {
  list-style: none;
  margin-left: 0;
  font-size: 2em;
  font-family: Helvetica, Arial, sans-serif;
  padding: 1em 0 1em 0;
  border-bottom-style: solid;
  border-color:#eeeeee;
  border-bottom-width: 0.05em;
}
```

This will put the list elements flush to the left and make the list occupy the entire screen. Now, when you swipe to the right on an iPhone TableView, the phone displays a **Delete** button to delete the current item. To approximate the button, here's another CSS rule:

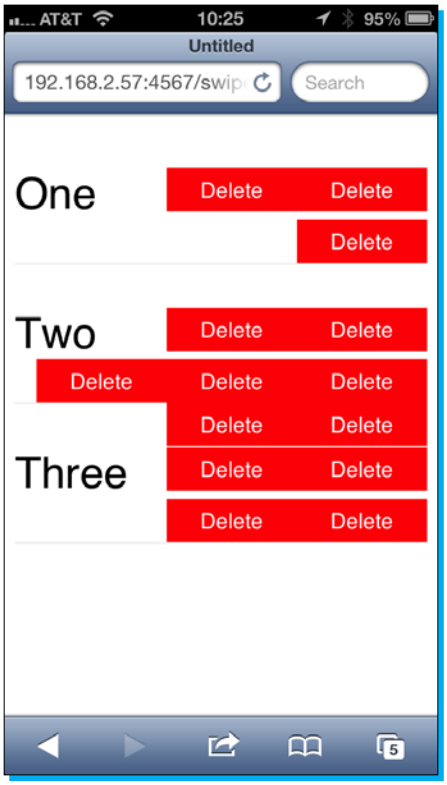
```
.delete-button {
  position: relative;
  background-color: red;
  color: white;
  height: 2em;
  width: 6em;
  font-size: 0.5em;
  float: right;
  line-height: 2em;
  text-align: center;
}
```

This won't look exactly like the button, though you could add further CSS rules to recreate the 3D and gloss effects that you see on a standard button as seen on iOS6. However, this will look like the flatter elements being introduced in iOS7. Making the button appear on a swipe event is fairly simple:

```
$(".list-element").on('swipe', function() {
  $(this).append("<button class='delete-button'>Delete
  </button>");
});
```

On every swipe event, the delete button will be inserted into the DOM inside the current list element. This works, but the button just appears violently from nowhere, does nothing, and hangs around when you swipe other list elements; moreover, if you swipe the same element again, you'll get another button, which is not the desired behavior.

Have a look at the following screenshot:



We can make the arrival of the delete button a little less jarring by adding an opacity rule to the button's CSS, then bringing it up to full opacity when swiped:

```
    $(this).append("<button class='delete-button'>Delete</button>");
    $('.delete-button').animate({'opacity': '1'});
```

Also, only one delete button should be visible at any one time. At this point, logic in the function being passed to the swipe event is starting to build up, so let's also split it out into another function and pass that into the event listener:

```
function swipe_to_delete() {
  $(".delete-button").remove();
  $(this).append("<button class='delete-button'>Delete</button>");
  $('.delete-button').animate({'opacity': '1'});
}
  $(".list-element").on('swipeRight', swipe_to_delete);
```

Having done all that, it's time to add the code that actually deletes an element. To do this, you'll add another event listener that looks out for a tap event on the delete button, and then delete the parent `lI` element.

```
function swipe_to_delete() {
  $(".delete-button").remove();
  $(this).append("<button class='delete-button'>Delete</button>");
  $('.delete-button').animate({'opacity': '1'});
  $(".delete-button").on('tap', function(){
    $(this).parent('li').remove();
  });
}
```

There is a slight problem with the code at this point, though; once you have swiped on one list item, you can't return to the state where there are no delete buttons on screen. This can be fixed by adding another listener to the list itself, watching out for a tap event to fade out and remove the current delete button.

```
function remove_delete_button() {
  $('.delete-button').animate(
    {'opacity': '0'}
  }
  $('ul').on('tap', remove_delete_button);
```

The final result will look something like this, after hooking into the `Zepto($)` function to bind all our event listeners at page load:

```
<!doctype html>
<html>
  <head>
    <meta name="viewport"
    content="width=device-width,user-scalable=no">
```

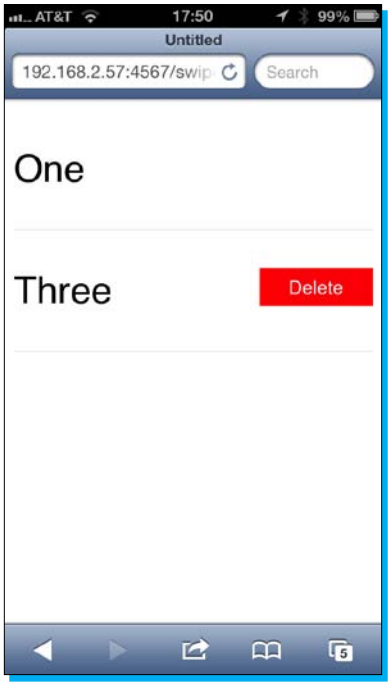
```
</head>
<body>
  <style>
    ul {
      width: 100%;
      margin: 0 0 0 0;
      padding: 0 0 0 0;
    }
    .list-element {
      list-style: none;
      margin-left: 0;
      font-size: 2em;
      font-family: Helvetica, Arial, sans-serif;
      padding: 1em 0 1em 0;
      border-bottom-style: solid;
      border-color:#eeeeee;
      border-bottom-width: 0.05em;
    }
    .delete-button {
      position: relative;
      background-color: red;
      color: white;
      height: 2em;
      width: 6em;
      font-size: 0.5em;
      float: right;
      opacity: 0.0;
      line-height: 2em;
      border: none;
    }
  </style>
  <div id="container">
    <ul>
      <li class="list-element">One</li>
      <li class="list-element">Two</li>
      <li class="list-element">Three</li>
    </ul>
  </div>
  <script src="zepto-touch.js"/></script>
  <script>Zepto(function($){
    $(".list-element").on('swipe', swipe_to_delete);
    $('ul').on('tap', remove_delete_button);
    function swipe_to_delete() {
      $(".delete-button").remove();
      $(this).append("<button class='delete-button'>Delete
      </button>");
      $('.delete-button').animate({'opacity': '1'});
      $(".delete-button").on('tap', function(){
        $(this).parent('li').remove();
      });
```

```
        }
        function remove_delete_button() {
          $('.delete-button').animate(
            {'opacity': '0'},
            { complete: function(){$(".delete-button").remove();}});
        }
      });
      </script>
    </body>
  </html>
```

The preceding example is also available at `https://github.com/falloutdurham/zeptojstarter`

While there would be more work remaining before this becomes a full interface (for example, providing a way of adding elements), you can see that, with fewer than 20 lines of JavaScript and some CSS rules, you can create a much richer experience than just a plain unordered list. Using other gestures, such as `swipeUp` and `swipeDown`, you could implement list reordering, pull-down hidden menus, or other interactions. These can be combined with the single or double tap events and the pinch gestures to create an expressive interface without ever once having to run native code on the mobile device.

The following screenshot shows how it will appear then:



Checking for jQuery compatibility

Unfortunately, there isn't a sure-fire way to determine whether a jQuery plugin will work in Zepto.js. A reasonable approach is to get the source of the plugin and search within it to see if the plugin uses features that aren't present in Zepto.js, which were outlined in the previous section (popular culprits will be if the plugin uses jQuery animation features such as fadeIn/Out or the new Promises-supporting Ajax functions). If those are present, then you'll have to put in a request to the plugin author to see if they are willing to rewrite the plugin to support Zepto.js or attempt to convert it yourself.

In any event, if the plugin doesn't have explicit support, you will have to modify the final line so that it extends Zepto.js instead of jQuery. Where the final line of a plugin will normally look like the following line of code:

```
})(jQuery)
```

It should be changed to be:

```
})(Zepto)
```

Or even `})(Zepto || jQuery)` to keep jQuery support.

Once that's done, you can load your web app and check in the browser's console log for any further errors or incompatibilities.

Writing plugins

Eventually, you'll want to make your own plugins. As you can imagine, they're fairly similar in construction to jQuery plugins (so they can be compatible). But what can you do with them? Well, consider them as a macro system for Zepto.js; you can do anything that you'd do in normal Zepto.js operations, but they get added to the library's namespace so you can reuse them in other applications.

Here is a plugin that will take a Zepto.js collection and turn all the text in it to Helvetica font-family at a user-supplied font-size (in pixels for this example).

```
(function($){
  $.extend($.fn, {
    helveticaize: function( options ){
      $.each(this, function(){
        $(this).css({"font-family":"Helvetica",
        "font-size": options['size']+'px'});
      });
      return this;
    }
  })
})(Zepto || jQuery)
```

Then, to make all links on a page Helvetica, you can call `$("a").helveticaize()`. The most important part of this code is the use of the `$.extend` method. This adds the `helveticaize` property/function to the `$.fn` object, which contains all of the functions that Zepto.js provides.

Note that you could potentially use this to redefine methods such as `find()`, `animate()`, or any other function you've seen so far. As you can imagine, this is not recommended—if you need different functionality, call `$.extend` and create a new function with a name like `custom_find` instead. In addition, you could pass multiple new functions to `$.fn` with a call to `$.extend`, but the convention for jQuery and Zepto.js is that you only provide as few functions as possible (ideally one) and offer different functionality through passed parameters (that is, through `options`). The reason for this is that your plugin may have to live alongside many other plugins, all of which share the same namespace in `$.fn`. By only setting one property, you hopefully reduce the chance of overriding a method that another plugin has defined.

In the actual definition of the method that's being added, it iterates through the objects in the collection, setting the font and size (if present) for all the objects in the collection. But at the rest of the method it returns `this`. Why? Well, if you remember, part of the power of Zepto.js is that methods are chainable, allowing you to build up complex selectors and operations in one line. And thanks to `helveticaize()` returning `this` (which will be a collection), this newly-defined method is just as chainable as all the default methods provided. This isn't a requirement of plugin methods but, where possible, you should make your plugin methods return a collection of some sort to prevent breaking a chain (and if you can't, for some reason, make sure to spell that out in your plugin's documentation).

Finally, at the end, the `(Zepto || jQuery)` part will immediately invoke this definition on either the Zepto object or jQuery object. In this way, you can create plugins that work with either framework depending on whether they're present, with the caveat, of course, that your method must work in both frameworks.

Differences between jQuery and Zepto

Although Zepto.js was designed with the idea of being a library that works like jQuery, it's important to note that it is not 100 percent jQuery-compatible. Earlier, you've seen that jQuery animation functions such as `fadeOut()` and `slideIn()` are not included by default by Zepto.js. There are other functions that jQuery offers that Zepto.js doesn't, and Zepto has a few that jQuery doesn't provide:

- jQuery CSS selector extensions are not supported

See `http://api.jquery.com/category/selectors/jquery-selector-extensions` for a list of these selectors.

However, Zepto.js does provide an optional selector module which provides a small subset, though even with this additional support, you still can't do complex selector addition—things such as `ul:first(has:li) + ul:visible(contains("this")` will not work. Unless you have a pressing need, it's best to avoid using the selector extensions altogether.

- The `clone()` method—in jQuery, calling `clone()` will make a deep copy of the object that includes event-handlers and data elements. Zepto.js will not copy these when `clone()` is called, so you will have to set them up again if you want them to exist in the new object.
- The `data()` method—to get the full jQuery support, you'll need to build in the `data` module. Otherwise, you're limited to just storing strings in data elements.
- jQuery allows you to fire global events such as `$.event.trigger('my-event')` that are not bound to any part of the DOM. In Zepto.js, all events must be triggered on a DOM object.
- The `contents()` method doesn't exist in Zepto.js. In most cases, you can get away with `children()` instead (comment and text nodes will not be included as they are in `contents()`, however).
- As you've seen previously, Zepto.js restricts itself to CSS3 animations by default, meaning older jQuery code that makes heavy use of jQuery-specific animation methods will have to be converted to CSS (which is probably a good idea anyway, as browsers these days are optimized for such animations, even using dedicated graphics hardware for faster and smoother operation).

Zepto.js itself has a few functions that aren't present in jQuery. These are mainly convenience functions that operate on collections, such as `concat`, `forEach`, `indexOf`, `pluck`, `push`, and `reduce`. Unless you have a pressing need (or know that your web app is only going to ever run on Zepto.js), it's best to avoid these too; if you do need to move back to jQuery at some point in the future, you will spend less time rewriting your code to work on the other library.

There's a list of plugins that support both Zepto.js and jQuery in the next section.

People and places you should get to know

If you're hungry for more information about Zepto.js, here are some good places to start.

Official sites

- Homepage: `http://zeptojs.com`
- Source code: `https://github.com/madrobby/zepto`

Articles and tutorials

- The Essentials of Zepto.js—another trip through the basics: `http://net.tutsplus.com/tutorials/javascript-ajax/the-essentials-of-zepto-js/`
- How to build fast HTML5 mobile apps using backbone.js, zepto.js and trigger.io: `http://trigger.io/cross-platform-application-development-blog/2012/03/02/how-to-build-fast-html5-mobile-apps-using-backbone-js-zepto-js-and-trigger-io/`
- How Zepto.js was used to build a font specimen app, Greta Sans: `https://www.typotheque.com/blog/greta_sans_specimen_app`
- Porting plugins from jQuery to Zepto.js: `http://blog.pamelafox.org/2011/11/porting-from-jquery-to-zepto.html`

Blogs

- Thomas Fuchs, creator of Zepto.js, often talks about Zepto.js milestones and new features on his main site, `http://mir.aculo.us/`

Twitter

- Follow Zepto.js on Twitter: `http://twitter.com/#zeptojs`
- Zepto.js Lead developer, Thomas Fuchs: `https://twitter.com/thomasfuchs`
- Zepto.js Second lead developer, Mislav Marohnić: `https://twitter.com/mislav`

GitHub

- Code samples for this book: `https://github.com/falloutdurham/zeptojstarter`
- PubSub example: `https://github.com/martinjuhasz/pubsub-zepto`
- Drag'n'Drop example: `https://github.com/rkusa/zepto-dnd`
- zProgress progress bar: `https://github.com/madrobby/zprogress`

Supported plugins and frameworks

Here's a non-exhaustive list of some plugins and frameworks that explicitly work with both Zepto.js and jQuery.

Frameworks

- Twitter Bootstrap: `http://twitter.github.io/bootstrap`
- Foundation 4: `http://foundation.zurb.com`
- Backbone.js: `http://backbonejs.org`

Although Zepto.js is supported, the maintainers of Backbone do not consider Zepto.js compatibility a high priority when updating the framework.

Plugins

- CanJS: `http://canjs.com/`
- Happy.js: `http://happyjs.com/`
- Garlic.js: `http://garlicjs.org/`
- Magnific Popup: `http://dimsemenov.com/plugins/magnific-popup/`
- iCheck: `http://damirfoy.com/iCheck/`
- jQT: `http://jqtjs.com/`
- Powertable: `http://trentrichardson.com/examples/jQuery-Powertable/`

For more Open Source information, follow Packt Open Source at `http://twitter.com/#!/packtopensource`.

Thank you for buying
Instant Zepto.js

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT]
PUBLISHING

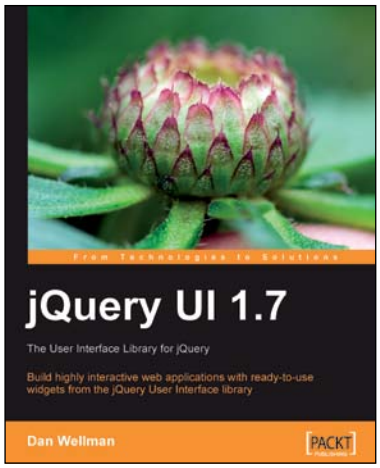
Ext JS 4 Web Application Development Cookbook

ISBN: 978-1-84951-686-0 Paperback: 488 pages

Over 110 easy-to-follow recipes backed up with real-life examples, walking you through the basic Ext JS features to advanced application design using Sencha's Ext JS

1. Learn how to build Rich Internet Applications with the latest version of the Ext JS framework in a cookbook style
2. From creating forms to theming your interface, you will learn the building blocks for developing the perfect web application
3. Easy to follow recipes step through practical and detailed examples which are all fully backed up with code, illustrations, and tips

---



jQuery UI 1.7: The User Interface Library for jQuery

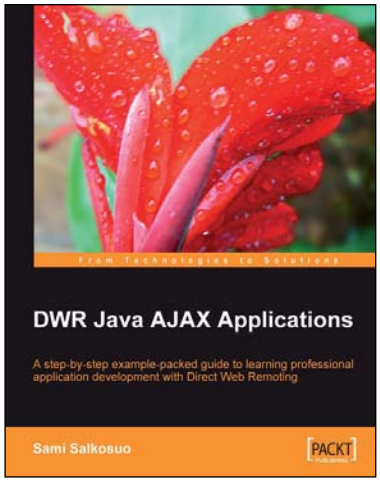
ISBN: 978-1-84719-972-0 Paperback: 392 pages

Build highly interactive web applications with ready-to-use widgets from the jQuery User Interface library

1. Organize your interfaces with reusable widgets: accordions, date pickers, dialogs, sliders, tabs, and more
2. Enhance the interactivity of your pages by making elements drag-and-droppable, sortable, selectable, and resizable
3. Packed with examples and clear explanations of how to easily design elegant and powerful front-end interfaces for your web applications

Please check **www.PacktPub.com** for information on our titles

[PACKT] PUBLISHING



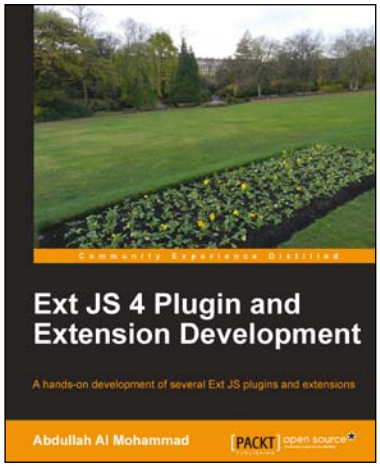
DWR Java AJAX Applications

ISBN: 978-1-84719-293-6 Paperback: 228 pages

A step-by-step example-packed guide to learning professional appliaction development with Direct Web Remoting

1. Learn Direct Web Remoting features from scratch and how to apply DWR practically
2. Topics such as configuration, testing, and debugging are thoroughly explained through examples
3. Demonstrates advanced elements of creating user interfaces and back-end integration

---



Ext JS 4 Plugin and Extension Development

ISBN: 978-1-78216-372-5 Paperback: 306 pages

A hands-on development of several Ext JS plugins and extensions

1. Easy-to-follow examples on ExtJS plugins and extensions
2. Step-by-step instructions on developing ExtJS plugins and extensions
3. Provides a walkthrough of several useful ExtJS libraries and communities