Extracted from:

Seven Mobile Apps in Seven Weeks

Native Apps, Multiple Platforms

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Seven Mobile Apps in Seven Weeks

Native Apps, Multiple Platforms

Tony Hillerson

Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

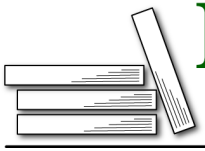
Seven Mobile Apps in Seven Weeks

Native Apps, Multiple Platforms

Tony Hillerson

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Candace Cunningham, Molly McBeath (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-148-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2016

Day 2: Making Time

Or, Managing the Clock List

Today we'll pick up where we left off yesterday. Today is about getting all the functionality of a world-clock app in place. We'll create an unstyled, vanilla UI that we can build on to add the mobile-specific features in Day 3.

For features, we'll get a default clock displaying the local time zone, display a list of time zones for users to choose from, and finish up by allowing users to edit the clock list by removing clocks they don't want to see. Let's get started.

Creating Clocks

As we've seen already, what drives the list of clocks is actually a list of time zones, which the clock interprets into a list of clocks using the current date of the browser. The time zone manager will manage a list of these time zones.

To have a clock to display initially, before the user chooses any clocks, we'll make sure that the first item in this list is the current time zone, based on the time provided by the browser.

Displaying the Current Time

Let's add a function to the time zone manager module to return the list of the user's saved time zones. We'll also have an option to prepend the browser's current time zone.

WebApp/web_02_01_default_clock/app/scripts/time_zone_manager.js

```
savedZones : function(includeCurrent) {
  var zones = [];
  if (includeCurrent) {
    var refDate = new Date();
    var offsetMinutes = refDate.getTimezoneOffset();
    zones.push({
      name: "Current",
      zone_name: "Current",
      offset: -offsetMinutes * 60,
      formatted_offset: this.formatOffsetMinutes(-offsetMinutes)
    });
  }
  return zones;
},

formatOffsetMinutes : function(offsetMinutes) {
  var offsetHours = offsetMinutes / 60;
  offsetHours = Math.abs(offsetHours).toString() + ":00";
```

```
  if (offsetMinutes < 600) offsetHours = "0" + offsetHours;
  if (offsetMinutes < 0) offsetHours = "-" + offsetHours;
  return offsetHours;
},
```

If the includeCurrent option is true, we'll construct a current time zone object like the ones the API returns.

With this code in place, we can test the time zone manager's savedZones() function in the console, like this:

```
> tzManager = $.app.namespaces.managers.TimeZoneManager
[Object {fetchTimeZones: function,
savedZones: function, formatOffsetMinutes: function}]
> tzManager.savedZones(true)
[Object
  formatted_offset: "-06:00"
  name: "Current"
  offset: -21600
  zone_name: "Current"
  __proto__: Object]
```

Now that we have a list of saved time zones to work with, we can render them on the page. First we need a way to have some code run when the page is completely loaded. We can use jQuery's ready for that, as we do here in main.js.

WebApp/web_02_01_default_clock/app/scripts/main.js

```
$(document).ready(function() {
  var tzManager = namespaces.managers.TimeZoneManager,
      clock = namespaces.models.Clock;
  tzManager.fetchTimeZones(function(timezones) {
    tzManager.createClocksIn($("#clockList"));
    clock.start();
  });
});
```

Now, when the page is loaded and ready for scripts to run, we tell the time zone manager to create clocks in the clock list and then tell the clock module to start updating the clocks on the screen by calling start().

Creating clocks, which is done here in the time zone manager (for now), is a matter of getting the saved zones and iterating through them with Underscore's each() function.

WebApp/web_02_01_default_clock/app/scripts/time_zone_manager.js

```
createClocksIn : function(list) {
  var zones = this.savedZones(true);
  _.each(zones, function(zone) {
    var item = $("<li class='clock'/>");
```

```
    $(list).append(item);
  });
}
```

For each zone, we create a new li tag with a class of clock, wrapped with jQuery, and then append that to the HTML list element passed to the function.

Next, the clock module's start() function is called.

WebApp/web_02_01_default_clock/app/scripts/clock.js

```
start: function() {
  this.tick();
  var tickFunction = _.bind(this.tick, this);
  setInterval(tickFunction, 1000);
},
```

This function fires the tick() immediately, so there's no rendering the clocks, but also sets up a JavaScript timer to fire every second using setInterval().

Pay attention to the use of Underscore's bind(). JavaScript has a rather unique and surprising approach to dynamic lexical scope. If you're new to JavaScript, at some point you'll find out the hard way that the this variable doesn't refer to what you think it does. When you're writing a JavaScript function, it seems reasonable to assume that this will point to something in the context where the function is defined, but that doesn't always hold true. The this variable always refers to the parent context of the function *at call time*.

Here, the tick() function, when the timer calls it, is going to be called by a different context. So your intuition may tell you that inside the tick() function this points to the Clock module, but actually it doesn't. If we need to call another function defined in clock module, we have to ensure that this points to the clock module. Underscore's bind function does this for us by either wrapping a native browser function or providing it for incompatible browsers. It returns a function where this is *bound* to the Clock module.

Now let's take a closer look at the tick() function.

WebApp/web_02_01_default_clock/app/scripts/clock.js

```
tick : function() {
  var date = new Date(),
      tzManager = namespaces.managers.TimeZoneManager,
      zones = tzManager.savedZones(true);
  var updateClockAtIndex = function(index, element) {
    var zone = zones[index],
        formattedTime = this.convertAndFormatDate(zone.offset, date);
    $(element).text(formattedTime);
  };
  updateClockAtIndex = _.bind(updateClockAtIndex, this);
```

```
  $(".clock").each(updateClockAtIndex);
},

convertAndFormatDate : function(offset, date) {
  var convertedSeconds = date.getUTCMinutes() * 60 +
        date.getUTCHours() * 3600 + offset,
      hour = Math.floor(convertedSeconds / 3600),
      minutes = Math.abs(
        Math.floor((convertedSeconds - (hour * 3600)) / 60)
      );
  if (hour < 0) {
    hour = hour + 24;
  } else if (hour >= 24) {
    hour = hour - 24;
  }
  var formattedTime = this.zeroPad(hour) + ":" + this.zeroPad(minutes);
  return formattedTime;
},

zeroPad : function(number) {
  var s = number.toString();
  var formattedNumber = (s.length > 1) ? s : "0" + s;
  return formattedNumber;
}
```

First we create a function that will update the time for a single time zone, given an index and an element from the clock list. To change the text of the list-item element, we wrap the element with jQuery and then call the text() function with the formatted time.

Just as with the tick() function before, we need to bind the function so this points to the clock module.

The bound function is then executed for each item returned by $(".clock") or each list item element where the class is clock.

Then we create a function, convertAndFormatDate(), for converting a date and offset to a time for display. Keep in mind that time zones around the world are not always offset from UTC by hours only, but sometimes by minutes as well. For instance, Chennai, in India, is UTC + 5:30, and Kathmandu, in Nepal, is UTC + 5:45.

Now we have the code in place to map a list of saved time zone objects onto a list of HTML list elements. Let's build out a way for users to choose which clocks they want to display.

Creating a Clock by Choosing a Time Zone

To allow users to add a new clock, we'll give them a link to click. We'll also add a new unordered list to hold the time zones.

WebApp/web_02_02_add_clock/app/index.html

```
<body>
  <a id="addClockLink">Add Clock</a>
  <ul id="zoneList"></ul>
  <ul id="clockList"></ul>
```

Then, to allow users to choose time zones, we'll need to respond to user input. This is a good opportunity to introduce a little better separation of concerns. For instance, right now the time zone manager module deals with creating clocks in the clock list. It would be better if we had a module to deal with the view and user input so that the time zone manager could focus on time zones.

This new file, a view controller module, will manage the view and be initialized from within main.js. Let's look at the steps to initialize the view controller.

We declare some new variables, some pointing to the HTML elements in the index page.

WebApp/web_02_02_add_clock/app/scripts/view_controller.js

```
var namespaces = $.app.namespaces,
    clock = namespaces.models.Clock,
    timeZoneManager = namespaces.managers.TimeZoneManager,
    clockList = $("#clockList"),
    zoneList = $("#zoneList"),
    addClockLink = $("a#addClockLink");
```

Then, the initialize function does some important setup.

WebApp/web_02_02_add_clock/app/scripts/view_controller.js

```
initialize: function() {
  this.openZoneListFunction = _.bind(this.addClockClicked, this);
  this.closeZoneListFunction = _.bind(this.dismissZoneList, this);
  addClockLink.click(this.openZoneListFunction);
  zoneList.hide();
  this.refreshClockList();
  clock.start();

  timeZoneManager.fetchTimeZones();
},
```

First we set some variables on the module pointing to bound versions of two functions, addClockClicked() and dismissZoneList(). These functions will handle when the Add Clock link is clicked and when the list of time zones is dismissed.

The addClockLink variable points to a jQuery element wrapping the anchor tag we just added to the HTML. Here we use jQuery's click() to set a click handler, so addClockClicked() is called when the link is clicked.

We then hide the list of time zones with jQuery's hide() function. Next we call refreshClockList(), which will build a list of clocks for each saved time zone in the time zone manager. Then we start the clock module ticking and fetch the list of time zones from the API.

When the Add Clock link is clicked, we present a list of time zones for the user to choose from. First we check to see if there are any children of the list. If there aren't, we assume they need to be created.

WebApp/web_02_02_add_clock/app/scripts/view_controller.js

```
addClockClicked : function() {
  if (zoneList.children().length === 0) {
    var zones = timeZoneManager.allZones();
        clickHandler = _.bind(this.zoneClicked, this);
    _.each(zones, function(zone, index) {
      var item = $("<li class='zone'/>");
      item.data("zoneIndex", index);
      item.text(zone.name);
      item.click(clickHandler);
```

```
      zoneList.append(item);
    });
  }
  this.presentZoneList();
},
```

To create the zone items, we create a bound function that points to zoneClicked(). Next we iterate through the list of all zones and create a list item for each zone. The text of the item will be the name of the zone, the click handler will be the bound function, and the item will be appended to the zone list.

The line where we call data() on the item uses a feature of jQuery that allows us to store arbitrary data on an element. We'll use this to store the index of the zone in a key called zoneIndex. That way, when an item is clicked and the click handler is fired, we'll be able to get the index from the data of the clicked item to know which time zone to add. Here's the code for the click handler.

WebApp/web_02_02_add_clock/app/scripts/view_controller.js

```
zoneClicked : function(event) {
  var item = $(event.currentTarget),
      index = item.data("zoneIndex");
  timeZoneManager.saveZoneAtIndex(index);
  this.dismissZoneList();
  this.refreshClockList();
},
```

First we grab the currentTarget of the incoming JavaScript click event and wrap it using jQuery. This will be the list item. Then we grab the data from that element, which tells us the associated zone index. We then tell the time zone manager to save the zone at that index. Finally we dismiss the zone list and refresh the list of clocks.

The code for saving time zones is rudimentary at this point, simply finding the zone at the given index and adding it to the list of saved time zones.

WebApp/web_02_02_add_clock/app/scripts/time_zone_manager.js

```
saveZoneAtIndex : function(index) {
  var zone = this.timeZones[index];
  this.savedTimeZones.push(zone);
},
```

Later we'll improve this code to store time zones in a more durable fashion, but this will get us through today.

Now we have a system in place for users to add as many clocks as they'd like to see in addition to the current time zone. Next let's look at managing the existing clock list.