CHAPTER 6

Improving the User Interface

In this chapter, you'll use the FCC Antenna Structure Registration (ASR) data you collected in Chapter 5 and create a mashup that really shines. What kind of interface surrounds a helpful map? What tricks can you do with a little more CSS and JavaScript? What kinds of things besides markers can you put on a map to increase its usefulness? You'll find some suggestions in this chapter.

This chapter begins where the middle of Chapter 5 left off, but if you're starting here, it's easy to catch up. We will use a controller much like we have for the previous chapters, together with a `map` view to display the map. We will continue to use the same project we've used throughout the book.

In this chapter, you'll learn how to use CSS and JavaScript to enhance your maps as follows:

- Have your map adjust its size to fill any browser
- Add a toolbar that hovers over the map
- Create side panels for your map
- Allow users to selectively view or hide groups of data points

Create a controller for your work in this chapter by typing **ruby script/generate controller chap_six** on the command line. Create a new `map` action on this controller with the code in Listing 6-1.

Listing 6-1. *map Action in app/controllers/chap_six_controller.rb*

```
def map
  @towers=Tower.find :all, :conditions=>['state = ? AND latitude < ?➥
     AND longitude > ?  AND latitude > ? AND longitude < ?',➥
     'HI', 20.40, -156.34, 18.52, -154.67]
end
```

Recall that in Chapter 5, we created the `Tower` model to represent antenna structures in the FCC ASR data. The `map` action utilizes the `Tower` model to retrieve all the antenna structures on Hawaii's Big Island. The numbers in the query represent the bounding latitude and longitude; the first latitude/longitude pair represents the upper left (northwest) corner, and the second latitude/longitude pair represents the lower right (southeast) corner. Note that we're also narrowing the query based on `state` being equal to `'HI'`; this is to speed up the query. If the database can restrict the result set to Hawaii towers only, it has a much smaller set of rows on which to perform the numeric comparisons to get the final result set.

CSS: A Touch of Style

CSS is the modern method of choice for controlling the visual appearance of an XML document. We'll put our CSS in a separate file: `public/stylesheets/style.css`.

In your `map` view, you'll need to add a reference to an external style sheet, as shown in Listing 6-2. Since its appearance will momentarily be controlled by this CSS file, it's also possible to remove the explicit size from the `map div`.

Listing 6-2. *views/chap_six/map.rhtml*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <script src="http://maps.google.com/maps?file=api&v=2&key=API_KEY"➥
type="text/javascript"></script>
     <%=javascript_include_tag 'prototype','application' %>
     <script type="text/javascript">
       var markers=<%=@towers.to_json%>;
     </script>
     <%=stylesheet_link_tag 'style'%>
</head>
<body id="body">
     <div id="map"></div>
</body>
</html>
```

Two other things are going on in this `map` view: we are including `prototype.js` via the `javascript_include_tag`, and we are outputting the markers in JSON format, just like we did in Chapter 5. We are including `prototype.js` because we are going to utilize some of its methods for the user interface in this chapter.

Without the `style` attribute, the `map div` collapses to nothing. Clearly, you need to actually *create* the style sheet and reapply the size declarations that were removed. Listing 6-3 shows the contents of the `public/stylesheets/style.css` file.

Listing 6-3. *public/stylesheets/style.css to Give the Map Dimensions*

```
#map {
    width: 500px;
    height: 400px;
}
```

■ **Tip** The # character in a style sheet means that you are defining an ID selector, so the `#map {...}` style will apply to the element with an ID of `map`, that is, `<div id="map"></div>`. On the other hand, if you want a style to apply to a *class* of elements, use the dot, which is the class selector. For example, `.my-class{...}` in your style sheet will apply to `<div class="my-class"></div>`. You can have multiple elements with the same class, and the style will apply to all of them. You'll spend most of your time with CSS writing either ID selectors, class selectors, or some combination thereof. To learn more about CSS, start with the tutorials at `http://www.w3schools.com/css/css_intro.asp`.

With the action, view, and CSS file in place, you just need an appropriate `application.js` to make sure your map initializes properly. The JavaScript code in Listing 6-4 should go in the usual `public/javascripts/application.js`. It contains the standard map initialization code you've seen before, utilizing the `markers` JavaScript variable to place points on the map.

Listing 6-4. *public/javascripts/application.js*

```
var map;
var centerLatitude = 19.6;
var centerLongitude = -155.5;
var startZoom = 9;
var markerHash={};
var currentFocus=false;

function addMarker(latitude, longitude, id) {
    var marker = new GMarker(new GLatLng(latitude, longitude));

    GEvent.addListener(marker, 'click',
        function() {
           focusPoint(id);
        }
    );

    map.addOverlay(marker);
    return marker;
}
```

```
function init() {
    map = new GMap($("map"));
    map.addControl(new GSmallMapControl());
    map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);
    for(i=0;i<markers.length; i++) {
      var current =markers[i];
      marker=addMarker(current.latitude, current.longitude,current.id);
      markerHash[current.id]={marker:marker,address:current.address,visible:true};
    }
}
window.onload=init;
```

In this code, note that we declare a global object called `markerHash` (the {} notation is equivalent to an empty object in JavaScript), and that we build up the object as we iterate through the `markers` array. We'll utilize this variable later, so just bear in mind when you see it next that we declared it here during initialization. Also, note that the `focusPoint` function is undefined at this point, which means that if you click a marker you will get a JavaScript error. We will define this function later in the section "Populating the Side Panel."

Going to `http://localhost:3000/chap_six/map`, you should see a map centered on Hawaii's Big Island, populated with 80 or 90 markers. Now, let's look at some CSS-based enhancements to the map.

Maximizing Your Map

A surprising number of Google Maps projects seem to use fixed-size maps. But why lock the users into particular dimensions when their screen may be significantly smaller or larger than yours? It's time to meet the map that fills up your browser, regardless of its screen size. Try swapping out your `style.css` file for Listing 6-5.

Listing 6-5. *Style.css for a Maximized Map*

```
html, body {
     margin: 0;
     padding: 0;
     height: 100%;
}

#map {
     position: absolute;
     top: 0;
     left: 0;
     width: 100%;
     height: 100%;
}
```

As you can see in Figure 6-1, the map is now completely flexible and fills any size of browser screen.

Figure 6-1. *Our map fills up the browser at 800 × 600.*

This method is particularly ideal for situations where a map is being used as part of a slide show or on a kiosk. However, it also works in the web page context, especially when combined with the trick described in the next section.

■ **Tip** Once you have the map maximized, you might notice how Internet Explorer 6 likes to show a disabled vertical scrollbar on our perfectly fitted page. Under most circumstances, this is actually desired behavior, since it means that centered sites are consistent with both short and long content. In our case, however, you really don't want it there. Fortunately, banishment is achieved with a pretty straightforward rule: `html { overflow: hidden; }`.

Adding Hovering Toolbars

CSS's `position` declaration opens up a world of options for styling your web pages. Using `position`, it's possible to layer multiple elements on top of one another, including text, images, and even scrolling Flash movies and scrolling `div` elements.

For the map, this means you can make content of various kinds hover on top of the map that the API generates. For comparison, Windows Live Local uses a full-screen map with translucent control widgets; check it out at `http://local.live.com/`.

Continuing the example from Listing 6-2, change your `map.rhtml` file to include some markup for a toolbar, as shown in Listing 6-6.

Listing 6-6. *map.rhtml with Added Markup for a Toolbar*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <script src="http://maps.google.com/maps?file=api&v=2&key=API_KEY"➥
type="text/javascript"></script>
     <%=javascript_include_tag 'prototype','application' %>
     <script type="text/javascript">
       var markers=<%=@towers.to_json%>;
     </script>
     <%=stylesheet_link_tag 'style'%>
</head>
<body id="body">
     <div id="map"></div>
     <div id="toolbar">
          <h1>Cell-Tower Locations</h1>
          <ul id="options">
               <li><a href="#">Towers</a></li>
               <li><a href="#">Poles</a></li>
               <li><a href="#">Masts</a></li>
               <li><a href="#">Other</a></li>
          </ul>
     </div>
</body>
</html>
```

And now, some CSS magic to take that markup and pull the toolbar up on top of the map. Add the styles in Listing 6-7 to your `style.css` file.

Listing 6-7. *Styles for a Floating Toolbar*

```
#toolbar {
     position: absolute;
     top: 20px;
     left: 60px;
     width: 400px;
     padding: 5px;
     background: white;
     border: 1px solid black;
}
```

You can see in Figure 6-2 that we've added a few more styles to make the toolbar's menu and titles prettier, but they're not critical to the layout example here. The important thing to note is the `position: absolute` bit. A block-level element such as a `div` naturally expands to fill all of the width it has available, but once you position it as absolute or float it, it no longer exhibits that behavior. So, unless you want it shrink-wrapping its longest line of text, you'll need to specify a width as either a fixed amount or some percentage of the window width.

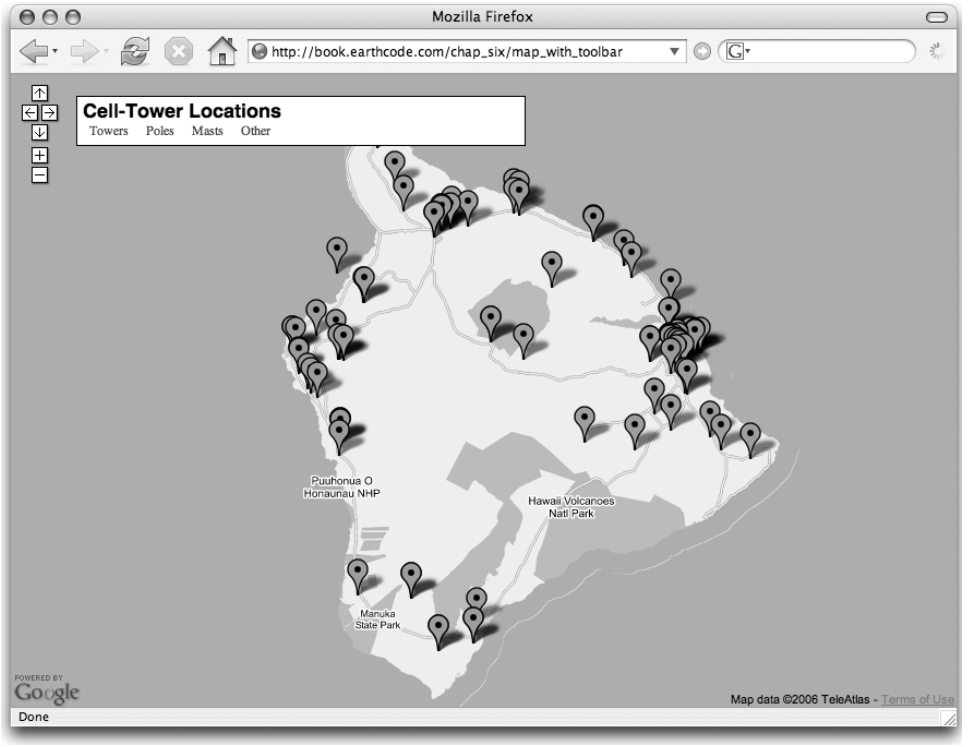

Figure 6-2. *Some styles for the toolbar*

WHAT ABOUT A FULL-WIDTH TOOLBAR?

Shouldn't it be possible to create a bar that's some fixed amount *less* than 100% of the available width? What about a floating toolbar that starts exactly 60 pixels from the left edge and then goes to exactly 40 pixels from the right edge?

It's possible in two different ways. You will see how to accomplish sizing maneuvers such as this using JavaScript. However, you can also create a full-width toolbar using just CSS. It's a little hairy, but there's certainly convenience (and possibly some pride, too) in keeping the solution all CSS.

The gist of the approach is that you need to "push in" the width of the absolutely positioned toolbar so that when it has a declared width of 100%, the 100% is 100% of the exact width you want it to have, rather than 100% of the browser's entire client area.

The toolbar `div` will need an extra wrapper around it, to do the "pushing in." So start by changing your markup:

```
<div id="toolbar-wrapper">
     <div id="toolbar">
     ...
     </div>
</div>
```

Now add the following styles to the `style.css` file:

```
#toolbar-wrapper {
     margin-right: 100px;
     position: relative;
}

#toolbar {
     width: 100%;
     ...
}
```

The right margin on the toolbar wrapper causes the toolbar itself to lose that horizontal space, even though the toolbar is ultimately being sucked out of the main document flow with `position: absolute`.

Creating Collapsible Side Panels

A common feature on many Google Maps mashups is some kind of side panel that provides supplementary information, such as a list of the pins being displayed. You can implement this simple feature in a number of ways. Here, we'll show you one that uses a little CSS and JavaScript to make a simple, collapsible panel.

First, the new side panel will need some markup. Modify the body section of Listing 6-2 to look like Listing 6-8.

Listing 6-8. *Index Body with Added Markup for a Side Panel*

```
<body id="body">
     <div id="map-wrapper">
          <div id="map"></div>
     </div>
```

```
    <div id="toolbar">
        ...
    </div>
    <div id="sidebar">
        <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin
        accumsan condimentum dolor. Vestibulum ante fabicum...</p>
    </div>
</body>
```

Notice that we have added two new `div` elements to the markup: `map-wrapper` and `sidebar`. In our style sheet, we can reference these two elements with ID selectors: `#map-wrapper` and `#sidebar`. To style this, you use almost the same trick as for the floating toolbar. This time, that wrapper has a margin that pushes the `map` `div` out of the way, so the elements appear beside each other, rather than overlapping. Listing 6-9 shows the CSS to add to `style.css`.

Listing 6-9. *New Styles for the Side Panel*

```
#map-wrapper {
    position: relative;
    height: 100%;
}

#sidebar {
    position: absolute;
    top: 0;
    width: 300px;
    height: 100%;
    overflow: auto;
}

#map-wrapper { margin-right: 300px; }
#sidebar {
    right: 0px;
    display:block;
}

body.sidebar-off #sidebar { display: none; }
body.sidebar-off #map-wrapper { margin-right: 0px; }
```

If you fill up the side panel with some more content, you can see how the `overflow` declaration causes it to scroll. It behaves just like a 1997-era frame, but without all the hassle of broken Back buttons and negative frame stigma.

■ **Note** Listing 6-9 provides only the simplest styles for this side panel. You'll find when you try to apply a right-side padding to `#sidebar` that it pushes in not just the content but also the scrollbar, which is an undesirable effect. Fortunately, it's an easy fix: just nest a `sidebar-contents` `div` inside the main side panel, and then put your styles on that. Alternatively, you can use the CSS selector `#sidebar p` to give special margins to all paragraphs residing inside.

Notice that we've defined two sets of styles for `#sidebar` and `#map-wrapper` in our CSS. The first set of styles is the default, which will be used in the absence of any more-specific CSS rules. The second set of styles (in bold in the listing) apply when the `body` element in the document has the `sidebar-off` class applied to it. The name of the game here is to effect multiple changes in your document by applying a single class change to the document body. In this case, when the `sidebar-off` class is applied to the body, the CSS effects two visible changes: the `sidebar` element is no longer displayed (its `display` property is set to `none`); and the `map-wrapper` element expands to the right (by changing its `margin` from `300px` to `0px`).

The effect is demonstrated in Figure 6-3, where we use the Firefox DOM Inspector to change the `body` element's `class` attribute, and suddenly the side panel vanishes. This example is rather trivial, since there are only two changes taking place. However, imagine a scenario in which you want many subtle style changes, depending on whether the side panel is visible. Controlling these changes through CSS rules is an easy way to manage the interface, even as the interface becomes more complex.

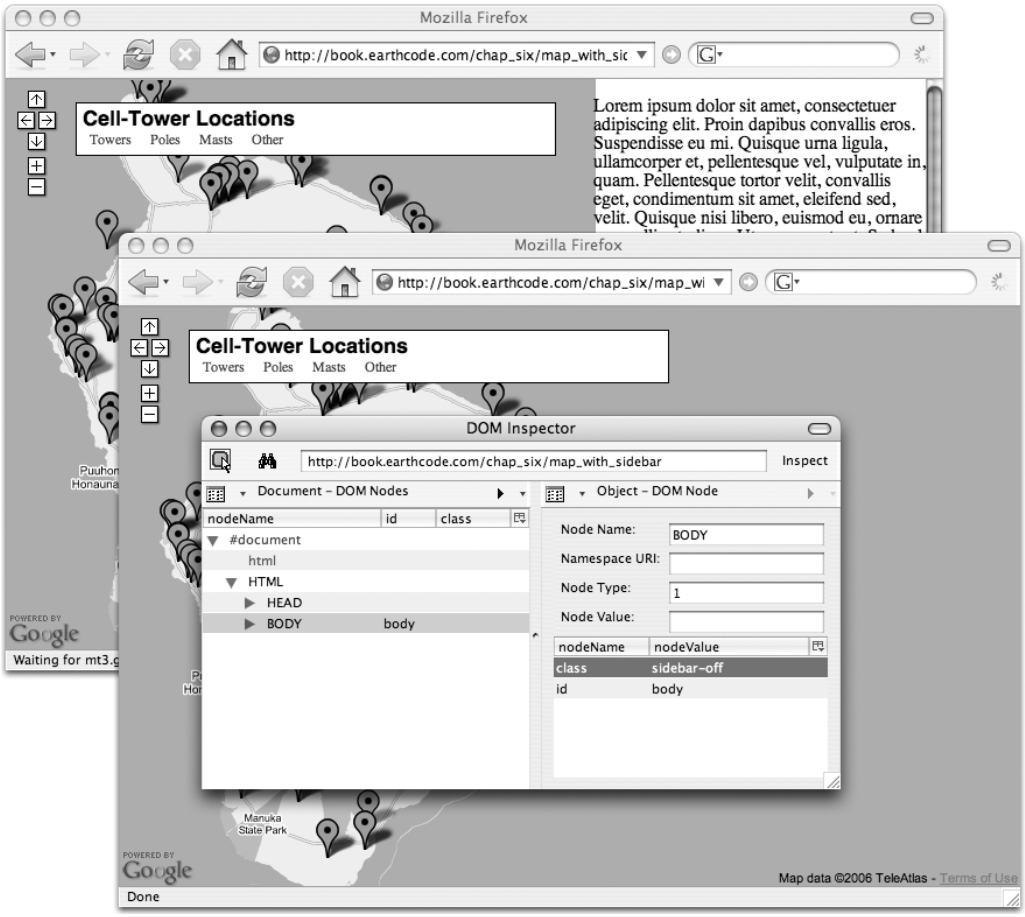

Figure 6-3. *The side panel obeys the body's class.*

Scripted Style

With the examples of the previous section in mind, we'll now examine a few ways to augment those CSS tricks with a little JavaScript.

Switching Up the Body Classes

You've seen that changing the body's `class` attribute can effect multiple changes in the interface. Now you just need to change the class through JavaScript triggered by a user event. To change the class, use Prototype's handy `addClassName` and `removeClassName` methods. You can embed the function calls in links in the toolbar, as shown in Listing 6-10.

Listing 6-10. *Side Panel Controls Added to map.rhtml*

```
<div id="toolbar">
    ...
  <ul id="sidebar-controls">
    <li><%=link_to_function 'hide',"Element.addClassName('body','sidebar-off')",➥
{:id=>'button-sidebar-hide'}%></li>
    <li><%=link_to_function 'show',"Element.removeClassName('body',➥
'sidebar-off')",{:id=>'button-sidebar-show'}%></li>
  </ul>
</div>
```

The `link_to_function` JavaScript helper takes a snippet of JavaScript code and attaches it to the `click` event of the link. In this case, the JavaScript snippet is a call to Prototype's `Element.addClassname` or `removeClassName` methods. The last argument to `link_to_function` is a hash of options for the HTML element the helper generates. We're just setting the ID of the resulting link here, but if you want to specify a `class` or a `style`, you would do so in this hash as well.

One important note: the first argument to `Element.addClassName` and `Element.removeClassName` is the ID of the element to change. It does *not* take the tag name, as our example may appear to communicate. Our code works because we have given the body tag `id=body` (refer back to Listing 6-2 to see where the ID is set).

ALTERNATIVES FOR ATTACHING EVENTS

In our toolbar, we use the built-in Rails `link_to_function` JavaScript helper to attach events to the show and hide links. There are alternative ways to attach events, including the following:

- Attach them directly in your JavaScript file: `$('button-sidebar-hide').onclick = function() {Element.addClassName('body','sidebar-off')};`.
- Utilize RJS (Rails JavaScript) templates and callbacks to move more of the logic server-side, thereby minimizing the amount of JavaScript in your application.
- Utilize the Unobtrusive JavaScript plug-in for Rails (`http://www.ujs4rails.com/`) to retain the convenience of the `link_to_function` helpers, but also to keep the JavaScript separate from the document markup.
- Utilize alternative JavaScript libraries (such as Behavior or jQuery) that are well-suited for attaching events unobtrusively.

The way we're doing it here (using the inline `link_to_element` JavaScript helper) is probably the most popular approach for Rails programmers, by virtue of it being the Rails default. The downside to this approach is that it muddies the resulting HTML with inline JavaScript code and it scatters your JavaScript event handlers between your JavaScript file and your `.rhtml` file(s). In many environments other than Rails, those would be considered significant downsides.

The approach you use will probably depend on the richness of your application's JavaScript-based functionality. Our experience has been that map-based applications are necessarily very rich in JavaScript code, simply because the Google Maps API itself is JavaScript. Once you have a large part of your application in JavaScript (the map handling code), it tends to change the center of gravity of your application and you use the inline JavaScript helpers and RJS less than you would otherwise. Your experience—and preferences—may vary, which is one of the reasons we're demonstrating the `link_to_function` approach in this example.

Finally, you can add some styles to spruce up the buttons a little. Using CSS, it's trivial to hide (or otherwise restyle) whichever button corresponds to the mode you're already in:

```
body.sidebar-off #button-sidebar-show { display: block; }
body.sidebar-off #button-sidebar-hide { display: none; }
```

Using these styles makes the two buttons appear to be the same one, as you can see in Figure 6-4.

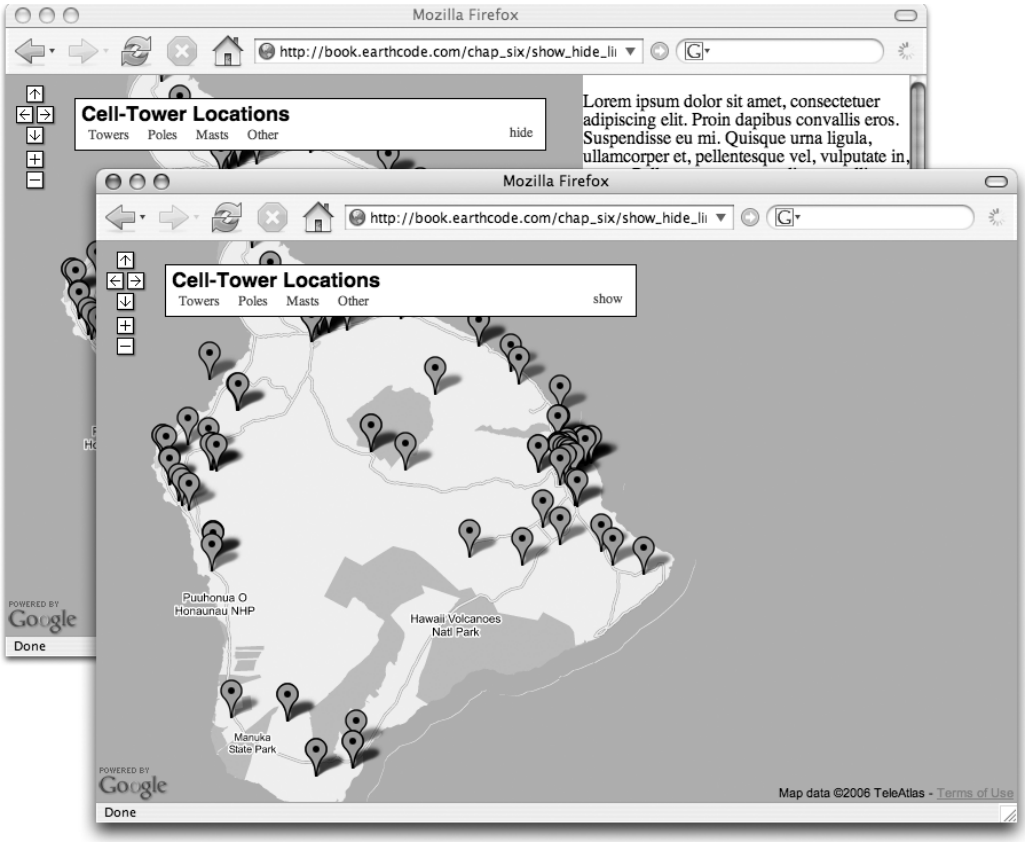

Figure 6-4. *The show/hide buttons behave as one, toggling the visibility of each other and of the side panel.*

Resizing with the Power of JavaScript

As you saw earlier, CSS gives you a significant amount of control over a page's horizontal layout. However, control over the vertical spacing is lacking. With only a style sheet, you can make a map the entire height of the window, or some percentage of that height, but you *cannot* make it be "from here to the bottom," or "100% minus 90 pixels." With JavaScript, however, this is very much possible.

JavaScript is an event-driven programming language. You don't need to be checking for things to happen all the time; you simply "hook" functionality onto various events triggered by the web browser.

With that in mind, all of the examples so far have already made use of the event `window.onload` to initialize the API and plot points on its map. What you're going to do next is hook some resizing functionality onto the event `window.onresize`. This code will execute when the window changes shape and resize the map to fit it.

Unfortunately, as is very obvious in the `windowHeight()` function of Listing 6-9, it has taken browser makers a long time to agree on how to expose the height of the client area to JavaScript. The method we've used here is the product of some exceptional research by Peter-Paul Koch (see `http://www.quirksmode.org/viewport/compatibility.html`). Incidentally, it's almost identical to the one Google itself uses to control the height of the Google Maps site's main map and side panel.

Pull up your `application.js` file and add the code shown in Listing 6-11 to it.

Listing 6-11. *Filling Vertical Space with the onresize Event*

```
function windowHeight() {
    // Standard browsers (Mozilla, Safari, etc.)
    if (self.innerHeight)
        return self.innerHeight;
    // IE 6
    if (document.documentElement && document.documentElement.clientHeight)
        return y = document.documentElement.clientHeight;
    // IE 5
    if (document.body)
        return document.body.clientHeight;
    // Just in case.
    return 0;
}

function handleResize() {
    var height = windowHeight();
    height -= document.getElementById('toolbar').offsetHeight - 30;
    document.getElementById('map').style.height = height + 'px';
    document.getElementById('sidebar').style.height = height + 'px';
}

function init() {
    ...

    handleResize();
}

window.onresize = handleResize;
```

The `handleResize()`function itself is actually pretty straightforward. The `offsetHeight` and `offsetWidth` properties are provided by the browser, and return—in pixels—the dimensions of their element, including any padding. Finding the correct height for the map and side panel

is simply a matter of subtracting that from the overall client window height, and then also removing the 30 pixels of padding that appear in three 10-pixel gaps between the top, the toolbar, the content area, and the bottom.

■ **Note** It's awkward to be individually assigning heights to the map and side panel. It would be cleaner if we could just assign the calculated height to a single wrapper and then set the children to each be permanently `height: 100%`. Indeed, such an approach works splendidly with Firefox. Unfortunately, Internet Explorer isn't able to get it quite right, so we're forced to use the slightly less optimal method of Listing 6-10.

Back in Listing 6-6, we placed the toolbar markup *after* the `map div` itself. This was partly arbitrary and partly because it's a convention to put the layers that are closer to the user later in the document. Now, however, the layering is to be removed in favor of a tiled approach, closer to what the Google Maps site itself uses. It's natural, then, to move the toolbar markup to before the map.

Also, we've added that `content` wrapper around the map and side panel. This is technically superfluous, but having a bit of extra markup to work with really helps to keep the style sheet sane. It's nearly always better to add wrappers to your template than to fill your CSS with ugly browser-specific hacks. (Some might disagree with us on this, but remember that wrappers are futureproof, while hacks can break with each new browser release.)

You can view the complete CSS changes that accompany Listing 6-12 on this book's web site, but it's not a dramatic departure from the styles of Listing 6-7. The changes are mostly aesthetic, now that the `handleResize()` method lets us do things such as put a nice 10-pixel margin between the key elements.

Listing 6-12. *Index Body with Markup Changes for Paneled Layout*

```
<body>
   <div id="toolbar">
          ...
   </div>
   <div id="content">
      <div id="map-wrapper">
        <div id="map"></div>
      </div>
      <div id="sidebar">
          ...
      </div>
   </div>
</body>
```

You can see how this example looks in Figure 6-5.

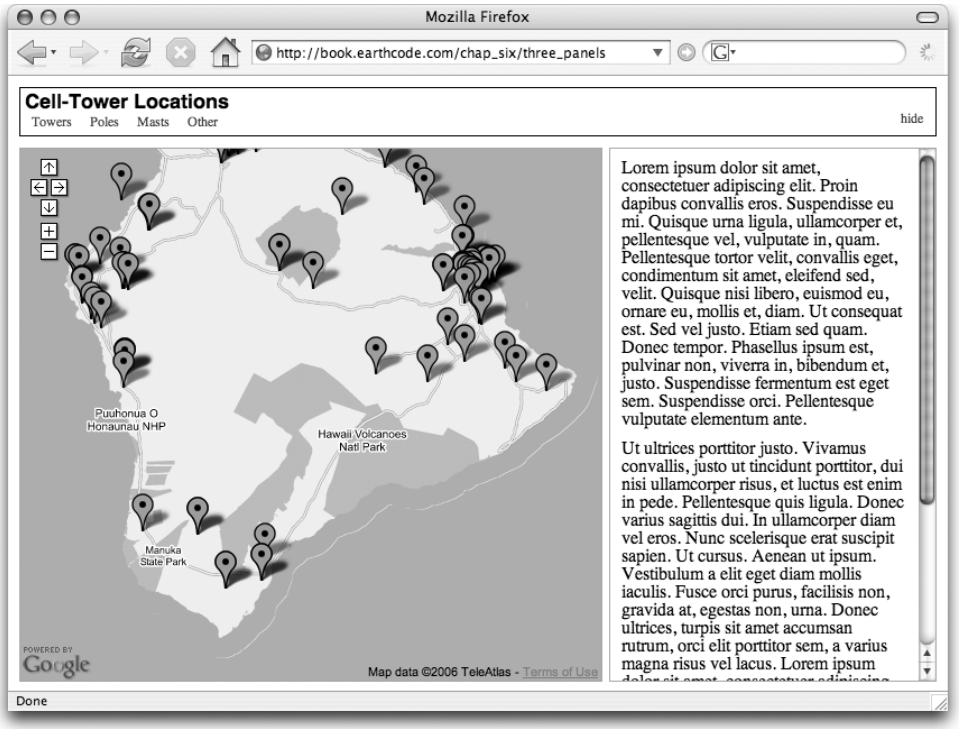

Figure 6-5. *The map area is divided into three elegant panels, one of which is collapsible.*

Populating the Side Panel

With our new side panel up and running, it would be good to get some actual content in there. A typical side panel use would be to present a list of all the markers plotted. This is particularly helpful when the markers are distributed in clusters. For example, a user could zoom in on an urban area to view a number of points bunched together, but she would be made aware that points exist elsewhere because that additional display has them listed.

To display the list of markers, you just need to edit the sidebar section of the `map.rhtml`, as shown in Listing 6-13.

Listing 6-13. *Generating the Side Panel in map.rhtml*

```
<div id="content">
    ...
  <div id="sidebar">
    <ul id="sidebar-list">
      <%@towers.each do |tower|%>
        <li id="sidebar-item-<%=tower.id%>">
          <%=link_to_function "<strong>#{tower.address}</strong> #{tower.city},➥
             #{tower.state} (#{tower.height}m", "focusPoint(#{tower.id})"%>
        </li>
      <%end%>
```

```
    </ul>
  </div>
</div>
```

Notice that our `link_to_function` is expecting a JavaScript function called `focusPoint`. Let's add that function to `application.js` now, as shown in Listing 6-14.

Listing 6-14. *The focusPoint Function in public/javascripts/application.js*

```
function focusPoint(id){
  markerHash[id].marker.openInfoWindowHtml(markerHash[id].address);
}
```

The function is just one line of code, but it's fairly dense. Recall that in the `init` function (which we defined back in Listing 6-4), we assembled an object called `markerHash`. Now, we're using `markerHash` to look up the current marker by ID, and calling the `openInfoWindowHtml` method on the marker we find. The `address` attribute (which we pass to `openInfoWindowHtml` for display) is also kept in the `markerHash` object for precisely this purpose. The key is that the `markerHash` contains both things we need: the `Marker` object itself, and the address to display in the info window.

The result of this will look similar to Figure 6-6. Note that we've enhanced the side panel with a few extra styles.

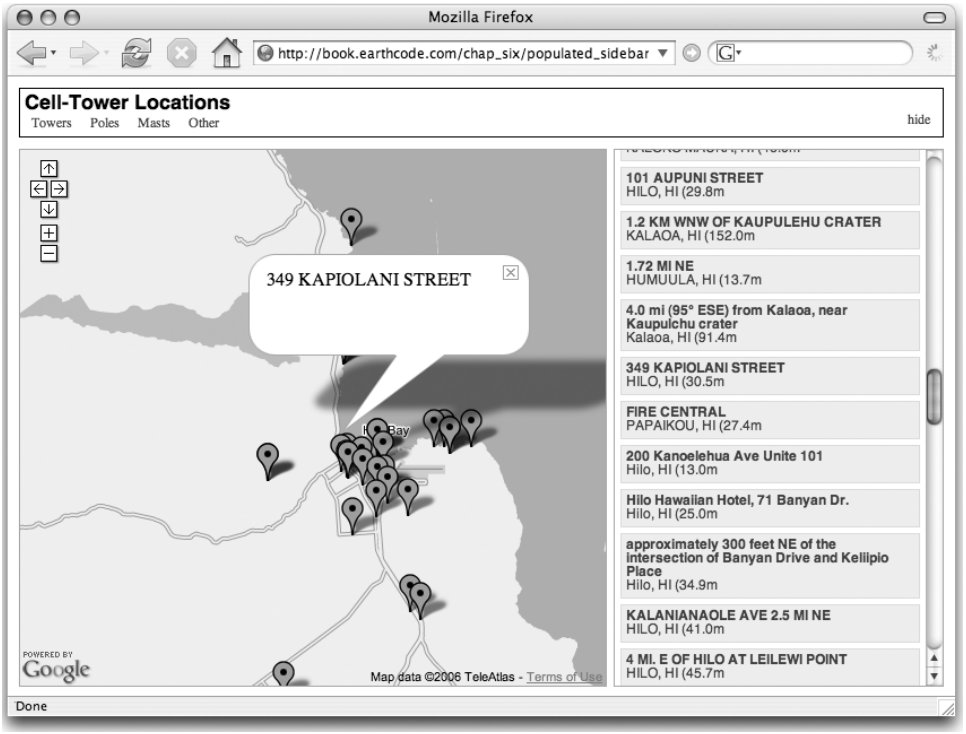

Figure 6-6. *The side panel populated with marker details*

Getting Side Panel Feedback

So far, users can interact with both the side panel item *and* the marker itself. However, they're receiving feedback through only the map marker—its info window pops up. It would be ideal if we could enhance this behavior by also highlighting the current point in the side panel list. Let's revisit the `focusPoint` function to provide this functionality. The code in Listing 6-15 should replace your current `focusPoint` function.

Listing 6-15. *An Enhanced focusPoint Function*

```
function focusPoint(id){
  if (currentFocus) {
    Element.removeClassName("sidebar-item-"+currentFocus,"current");
  }
  Element.addClassName("sidebar-item-"+id,"current");
  markerHash[id].marker.openInfoWindowHtml(markerHash[id].address);
  currentFocus=id;
}
```

The `currentFocus` global variable is the key to this problem. The first time the `focusPoint` function is called, `currentFocus` is `false` (see Listing 6-4 near the top to see where this variable is declared). On subsequent clicks, `currentFocus` is set to the ID of the previously clicked item. At that point, the `if` statement evaluates to `true`, and the previously focused item is defocused before applying focus to the newly clicked item.

Visually, *focus* means applying a class called `current` to a list item in the side panel. But first, we have to *remove* the `current` class from the last item selected. The `if` statement filters out the very first click when `currentFocus` isn't set yet.

And once again, with a few styles thrown in, you can see the results in Figure 6-7. The visual treatment in the side panel is subtle in this screenshot, but you can see that the border is stronger on the selected side panel item.

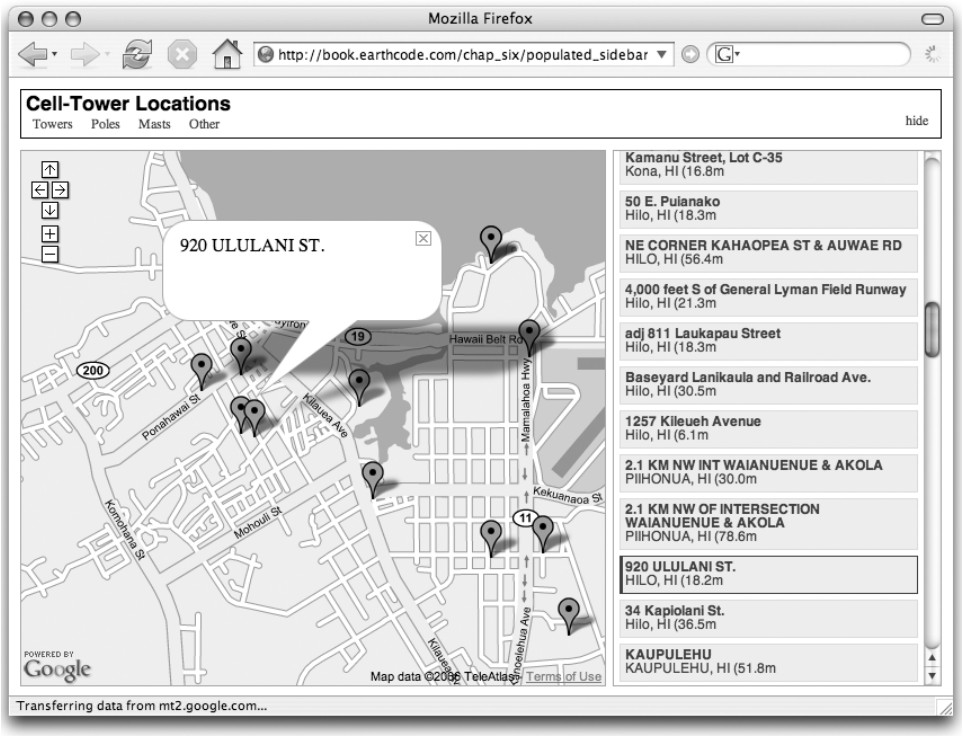

Figure 6-7. *The selected item in the side panel is highlighted.*

Data Point Filtering

One more area of the application still shows dummy content: the links underneath Cell-Tower Locations in the top toolbar. Right now these links are static. Let's make them reflect the actual structure types in the database and filter the map markers and side panel list when the links are clicked.

We're going to do the filtering browser-side, completely in JavaScript. Depending upon the application, this may or may not be the optimal approach. As an alternative, you may want to implement an Ajax callback to retrieve new side panel HTML and a new JSON data structure for the markers.

We are not making an Ajax call here; we're just using JavaScript to selectively limit what is displayed. When the entire data set for Hawaii is less than 40KB, what would be the point of breaking it up into multiple server calls? When you grab it in one big lump, it makes for a more seamless user interface since there's no waiting around for network latency on a 5KB file.

To provide the filtering mechanism, the first step is to identify the unique structure types in our dataset. To accomplish this, add the bold line of code in Listing 6-16 to your `app/controllers/chap_six_controller.rb` file.

Listing 6-16. *map Action in app/controllers/chap_six_controller.rb*

```
def map
  @towers=Tower.find :all, :conditions=>['state = ? AND latitude < ?➥
     AND longitude > ?  AND latitude > ? AND longitude < ?',➥
     'HI', 20.40, -156.34, 18.52, -154.67]

  @types=@towers.collect{|tower| tower.structure_type}.uniq
end
```

This line of code iterates through the `@towers` array, using the array's `collect` method to extract the `structure_type` from each item. Calling `uniq` on the result yields an array of unique structure types. We'll use this to render the links in the toolbar. Listing 6-17 shows the code in `app/views/chap_six/map.rhtml`.

Listing 6-17. *Showing @types in app/views/chap_six/map.rhtml*

```
<div id="toolbar">
  <h1>Cell-Tower Locations</h1>
  <ul id="filters">
    <li><%=link_to_function 'All', "filter('All')"%></li>
    <%@types.each do |type|%>
      <li><%=link_to_function type, "filter('#{type}')"%></li>
    <%end%>
  </ul>
 ...
```

In addition to iterating through the `@types` array and outputting links to the `filter` function, we are also outputting an All link, which will display all the structures in the dataset.

From here, we just need to implement the `filter()` function in JavaScript. Listing 6-18 shows the code to add to `public/javascripts/application.js`.

Listing 6-18. *The* `filter()` *Function in public/javascripts/application.js*

```
function filter(type){
  for(i=0;i<markers.length;i++) {
    var current=markers[i];
    if (current.structure_type == type || 'All' == type) {
```

```
      if (!markerHash[current.id].visible) {
        Element.show("sidebar-item-"+markers[i].id);
        map.addOverlay(markerHash[current.id].marker);
        markerHash[current.id].visible=true;
      }
    } else {
      if (markerHash[current.id].visible) {
        Element.hide("sidebar-item-"+markers[i].id);
        map.removeOverlay(markerHash[current.id].marker);
        markerHash[current.id].visible=false;
      }
    }
  }
}
```

Let's analyze what's going on in this function:

- The `for` loop iterates through all of the items in the `markers` array. Recall that the `markers` array was outputted as JSON in our view (see Listing 6-2).
- Inside the loop, there are just two possibilities: either we show the current marker or we hide the current marker. If the passed type is `all` we really don't want to discriminate; we'll show each marker regardless of its type.
- The `markerHash` object plays an important role, regardless of whether we are hiding or showing the current marker. Recall that `markerHash` was declared as a global variable and populated in the `init` method. If you look back at the population of `markerHash` (in Listing 6-4), you'll see that each entry contains a property called `visible`. At initialization time, each `visible` property was set to `true`. Once the loop determines whether the current marker should be shown, it uses the `visible` property to determine whether the marker is already displayed or not; after all, you don't want to add a marker to the map if it is already there.
- There are two steps to showing a marker. First, the list item in the side panel needs to be revealed, for which we use Prototype's `Element.show()` method. `Element.show()` takes a DOM element ID as a parameter, for which we use the marker's ID. Second, the marker itself needs to be added to the map, which we accomplish through the standard `map.addOverlay` method.
- The logic for removing markers is similar to the logic for showing markers but reversed.

The final result, with filtering, is shown in Figure 6-8.

Figure 6-8. *Marker filters in action*

You now have a reasonably full-featured map interface. You have an expandable full-page map, a collapsible side panel, side panel items that work in concert with the markers on the map, and, now, marker type filtering. You can use these techniques as a model for many different kinds of Google Maps mashups and applications.

RJS and Draggable Toolbars

One of the great things about Ruby on Rails is its flexibility. There are multiple ways of doing almost anything you need to do, and the functionality we've implemented in this chapter is no exception. Before we wrap up, let's look at two additional points to help you approach user interface challenges you may encounter.

RJS Templates and Partials

RJS is the JavaScript equivalent of RHTML: template files that are evaluated server-side through the Ruby interpreter before being served up to the browser. A common way of implementing the collapsible side panel and filtering is to utilize a `link_to_remote` callback to an RJS template, which would then effect the desired changes in the user interface. Is RJS a good choice for our application?

RJS would certainly be a reasonable choice. For this application, however, there would be a downside. RJS calls always involve a remote request (via XMLHTTP) back to the server, and in our application we can avoid the network latency by implementing the functionality directly in JavaScript.

The longer answer is that in our experience the amount of JavaScript coding necessarily involved with a Google Maps–based application changes the application's center of gravity, and you end up coding more in JavaScript. This is especially true as you make more sophisticated user interfaces, such as the JavaScript map resizing function we implemented in this chapter.

Other factors could change the equation. For example, if the filtering mechanism were more complex—perhaps a full-text search operation on marker names or data—it might make more sense to perform the filtering operation server-side and expose the interface updates in an RJS template.

Draggable Toolbars

Could you make the toolbar and/or side panel draggable? You certainly could—and with a minimal amount of effort—using the bundled Scriptaculous library. First, include all the default Rails JavaScript files with `<%=javascript_include_tag :defaults %>`. Next, include the following line in your view: `<%= draggable_element(:toolbar, :revert=>false) %>`. We aren't focusing on Scriptaculous here, but there are a lot of good examples available at `http://script.aculo.us/`.

Though it's relatively easy to make an element draggable, we caution you against doing it just because you can. It pays to ask whether "draggability" will actually enhance the user's experience. And if you do decide to implement it, think through the changes you will need to make to fully integrate the functionality. For example, if the top toolbar is draggable, what will happen to the space it occupies when you drag it away? Will the map and sidebar snap upward and expand vertically to take up the space? And what will happen when the toolbar is returned to the top? Will the map snap back down to give it space? Within what tolerance will the snap-back happen?

Remember also that draggable elements' positions don't automatically persist across page views. Which means that a toolbar dragged someplace will revert to its original place if the user refreshes the page. If that behavior confuses your users, you would probably have to implement some sort of persistence mechanism for the position, perhaps a behind-the-scenes Ajax call to store the new position in the session.

This isn't to discourage you from experimenting with features such as drag-and-drop, only to get you to think through the full effects on the user experience.

Summary

In this chapter, we took a look at a number of JavaScript and CSS techniques for making your maps better-looking and more functional. We made the map adjust its size to fill the browser window, and we added a toolbar that hovers over the map. We created a side panel for the map, and implemented JavaScript to show and hide the side panel. Finally, we created a JavaScript filtering mechanism to allow users to selectively hide and show groups of markers on the map.

In Chapter 7, you'll continue to develop this code, focusing on how to deal with the vastness of the full U.S.-wide FCC ASR database.