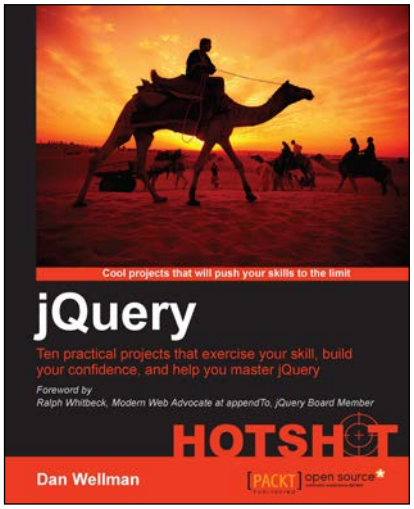
[PACKT] open source
community experience distilled
PUBLISHING

jQuery HOTSHOT

Dan Wellman



Chapter No. 3

"An Interactive Google Map"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "An Interactive Google Map"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Dan Wellman is an author and front-end engineer who lives on the South Coast of the UK and works in London. By day he works for Skype, writing application-grade JavaScript, and by night he writes books and tutorials focused mainly on front-end development. He is also a staff writer for the Tuts+ arm of the Envato network, and occasionally writes for .Net magazine. He's a proud father of four amazing children, and the grateful husband of a wonderful wife. This will be his seventh book.

I'd like to thank my family and friends for their continued support, you guys rock. I'd also like to thank my tireless PA, Derek Spacagna, for his persistent encouragement, and my friend Michael Chart, without whose mathematical genius some of the examples would not have been possible.

jQuery HOTSHOT

Welcome to *jQuery Hotshot*. This book has been written to provide as much exposure to the different methods and utilities that make up jQuery as possible. You don't need to be a jQuery hotshot to read and understand the projects this book contains, but by the time you've finished the book, you should be a jQuery hotshot.

As well as learning how to use jQuery, we are also going to look at a wide range of related technologies including using some of the more recent HTML5 and related APIs, such as localStorage, how to use and create jQuery plugins, and how to use other jQuery libraries such as jQuery UI, jQuery Mobile, and jQuery templates.

jQuery has been changing the way we write JavaScript for many years. It wasn't the first JavaScript library to gain popularity and widespread usage among developers, but its powerful selector engine, cross-browser compatibility, and easy-to-use syntax quickly propelled it to be one of the most popular and widely-used JavaScript frameworks of all time.

As well as being easy-to-use and abstracting complex and powerful techniques into a simple API, jQuery is also backed by an ever-growing community of developers, and is possibly the only JavaScript library protected by a not-for-profit foundation to ensure that development of the library remains active, and that it remains open source and free for everyone for as long as it's available.

One of the best things is that anyone can get involved. You can write plugins for other developers to use in order to complete common or not-so-common tasks. You can work with the bug tracker to raise new issues, or work with the source to add features, or fix bugs and give back in the form of pull requests through Git. In short, there is something to do for everyone who wants to get involved, whatever their background or skillset.

Getting started with jQuery

Every project in this book is built around jQuery; it's the foundation for everything we do. To download a copy of jQuery, we can visit the jQuery site at `http://jquery.com/`. There are download buttons here to obtain production and development versions of the library, as well as a wealth of other resources including full API documentation, tutorials, and much, much more to help you familiarize yourself with using the library.

One of the core concepts of jQuery is based on selecting one or more elements from the Document Object Model (DOM) of a web page, and then operating on those elements somehow using the methods exposed by the library.

We'll look at a range of different ways of selecting elements from the page throughout the projects in the book, as well as a wide selection of the different methods we can call on elements, but let's look at a basic example now.

Let's say there is an element on a page that has an `id` attribute of `myElement`. We can select this element using its `id` with the following code:

```
jQuery("#myElement");
```

As you can see, we use simple CSS selectors in order to select the elements from the page that we wish to work with. These can range from simple id selectors as in this example, class selectors, or much more complex attribute selectors.

As well as using jQuery to select elements, it is also common to use the `$` alias. This would be written using `$` instead of `jQuery`, as follows:

```
$("#myElement");
```

Once the element has been selected in this way, we would say that the element is wrapped with jQuery, or that it's a jQuery object containing the element. Using the `jQuery` (or `$`) method with a selector always results in a collection of elements being returned.

If there are no elements that match the selector, the collection has a length of `0`. When `id` selectors are used, we would expect the collection to contain a single element. There is no limit as to how many elements may be returned in the collection; it all depends on the selector used.

We can now call jQuery methods that operate on the element or elements that have been selected. One of the great features of most jQuery methods is that the same method may be used to either get a value, or set a value, depending on the arguments passed to the method.

So to continue our example where we have selected the element whose id attribute is `myElement`, if we wanted to find out its width in pixels, we could use jQuery's `width()` method:

```
$("#myElement").width();
```

This will return a number which specifies how many pixels wide the element is. However, if we wish to set the width of our element, we could pass the number of pixels that we'd like the element to have its width set to as an argument to the same method:

```
$("#myElement").width(500);
```

Of course, there is much more to using jQuery than these simple examples show, and we'll explore much more in the projects contained in this book, but this simplicity is at the heart of the library and is one of the things that have made it so popular.

What This Book Covers

Project 1, Sliding Puzzle, helps us build a sliding puzzle game. We'll use jQuery and jQuery UI together to produce this fun application and also look at the localStorage API.

Project 2, A Fixed Position Sidebar with Animated Scrolling, helps us implement a popular user interface feature – the fixed-position sidebar. We focus on working with the CSS of elements, animation, and event handling.

Project 3, An Interactive Google Map, teaches us how to work with Google's extensive Maps API in order to create an interactive map. We look at a range of DOM manipulation methods and look at how to use jQuery alongside other frameworks.

Project 4, A jQuery Mobile Single-page App, takes a look at the excellent jQuery Mobile framework in order to build a mobile application that combines jQuery with the Stack Exchange API. We also look at jQuery's official template engine, JsRender.

Project 5, jQuery File Uploader, uses jQuery UI once again, this time implementing a Progressbar widget as part of a dynamic front-end file uploader. We also cover writing jQuery plugins by making our uploader a configurable jQuery plugin.

Project 6, Extending Chrome with jQuery, shows us how we can extend the popular Chrome web browser with an extension built with jQuery, HTML, and CSS. Once again we make use of JsRender.

Project 7, Build Your Own jQuery, takes a look at how we can build a custom version of jQuery using a range of key web developer's tools including Node.js, Grunt.js, Git, and QUnit.

Project 8, Infinite Scrolling with jQuery, takes a look at another popular user-interface feature – infinite scrolling. We focus on jQuery's AJAX capabilities, again use JsRender, and look at the handy imagesLoaded plugin.

Project 9, A jQuery Heat Map, helps us build a jQuery-powered heat map. There are several aspects to this project including the code that captures clicks when pages are visited, and the admin console that aggregates and displays the information to the site administrator.

Project 10, A Sortable, Paged Table with Knockout.js, shows us how to build dynamic applications that keep a user interface in sync with data using jQuery together with the MVVM framework Knockout.js.

Project 3

An Interactive Google Map

In this project we'll create a highly interactive Google map that works with the latest version of Google's API to produce a map with custom overlays and markers, geocoded addresses, and computed distances. We'll also look at how to keep our simple UI in sync with the locations added to the map using a combination of Google and jQuery event handlers.

Mission Briefing

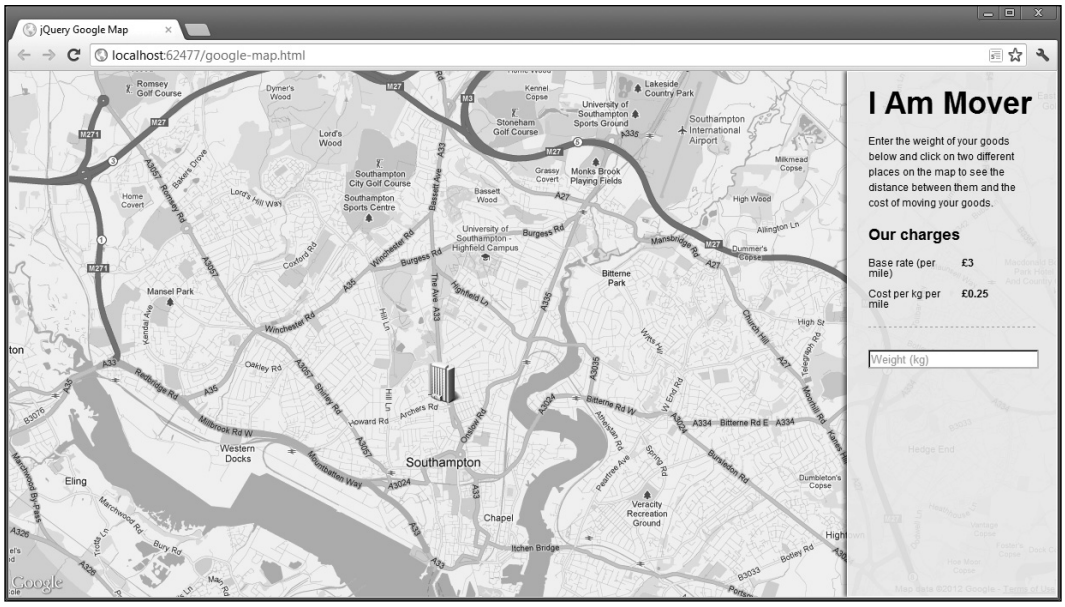
For the purposes of this project, we'll have a scenario where we need to build a map-based application for a company that takes things from one place to another. They want a page that their customers can visit to calculate the cost of, and maybe order, the transport of something from one place to another by clicking on different areas of a localized map.

We'll see how to listen for clicks on the map so that markers can be added and the precise locations of each marker can be recorded. We can then update the UI to show the actual street addresses of the locations that were clicked and allow the visitor to generate a quote based on the computed distance between the two addresses.

Why Is It Awesome?

Google Maps is a fantastic API to build on. Already highly interactive and packed with features, we can build robust and highly functional applications on top of the solid foundation it provides. Google provides the mapping data and interactivity with the map, while jQuery is used to build the UI – a winning combination.

The page that we'll end up with will resemble the following screenshot:



Your Hotshot Objectives

This project will be broken down into the following tasks:

- Creating the page and interface
- Initializing the map
- Showing the company HQ with a custom overlay
- Capturing clicks on the map
- Updating the UI with the start and end locations
- Handling marker repositions
- Factoring in weights
- Displaying the projected distance and cost

Mission Checklist

We'll need to link to a script file provided by Google in order to initialize the map and load in the API. We can also create the new files that we'll be using in the project at this point.

Don't worry, we don't need an API key from Google or anything like that for this project to work, we can just use the script by linking directly to it.

The Google Maps API is feature-rich and stable, and contains entry points for all of the best known mapping features, including Street View, geolocation, and the directions service. As well as the configuration options we used here, there are many, many others. For further information, see the documentation site at `http://developers.google.com/maps/`.

First we should save a new copy of the template file to our root project folder and call it `google-map.html`. Also create a `google-map.css` file and a `google-map.js` file and save them in the `css` and `js` folders respectively.

We can link to the style sheet for this example by adding the following `<link>` element to the `<head>` of the page, directly after the `<link>` element for `common.css`:

```
<link rel="stylesheet" href="css/google-map.css" />
```

Don't forget, we're using `common.css` with each project so that we can focus on the styles we actually need for the project, without all of the boring reset, float-clears, and other common CSS styling required for most web pages.

We can link to Google's script file, as well as the JavaScript file we just created, using the following `<script>` elements, directly after the `<script>` element for jQuery:

```
<script src="http://maps.googleapis.com/maps/api/js?sensor=false">
</script>
<script src="js/google-map.js"></script>
```

We'll also be using a couple of images in this project, `hq.png` and `start.png`, which can both be found in the accompanying code download for this book. You should copy them into the `img` directory in your local `jquery-hotshots` project directory. Our page is now set up ready for the first task.

Creating the page and interface

In our first task we can add the different containers for the map, and the initial UI elements needed by the page. We can also add some basic styling to lay things out as we want.

Engage Thrusters

We should add the following elements to the `<body>` element in the `google-map.html` page that we just set up:

```
<div id="map"></div>
<div id="ui">
    <h1>I Am Mover</h1>
    <p>Enter the weight of your goods below and click on two
    different places on the map to see the distance between
    them and the cost of moving your goods.</p>
    <h3>Our charges</h3>
    <dl class="clearfix">
        <dt>Base rate (per mile)</dt>
        <dd>&pound;3</dd>
        <dt>Cost per kg per mile</dt>
        <dd>&pound;0.25</dd>
    </dl>
    <input id="weight" placeholder="Weight (kg)" />
</div>
```

For some basic styling and to lay out the page ready for when we initialize the map, we can add the following selectors and styles to the `google-map.css` file that we just created:

```
#map { width:100%; height:100%; }
#ui {
    width:16%; height:99.8%; padding:0 2%;
    border:1px solid #fff; position:absolute; top:0; right:0;
    z-index:1; box-shadow:-3px 0 6px rgba(0,0,0,.5);
    background-color:rgba(238,238,238,.9);
}
#ui h1 { margin-top:.5em; }
#ui input { width:100%; }
#ui dl {
    width:100%; padding-bottom:.75em;
    border-bottom:1px dashed #aaa; margin-bottom:2em;
}
#ui dt, #ui dd { margin-bottom:1em; float:left; }
#ui dt { width:50%; margin-right:1em; clear:both; }
#ui dd { font-weight:bold; }
```

Objective Complete - Mini Debriefing

In this task we're just getting started by adding the underlying HTML elements that we'll populate properly over the next few tasks. A slightly boring, but somewhat necessary, first step in getting the example page up and running, and the project under way.

The first element we added is the container that the Google Maps API will render the map tiles into. We give it an `id` of `map` so that it can be efficiently selected, but it is completely empty to start with.

The next element is the container for the various UI elements the example requires. It too has an `id` of `ui` for easy selecting from our script, as well as for adding the CSS styling with.

Styling with IDs

Avoiding the use of ID selectors to add CSS styling is well on its way to becoming a general best practice, with tools such as **CSSLint** advising against its use.

While the arguments for doing this and sticking to classes, element, or attribute selectors are compelling, we'll be working with them in some of the projects throughout this book for simplicity.

CSSLint is an open source CSS code quality tool that performs static analysis of source code and flag patterns that might be errors or otherwise cause problems for the developer. See `http://csslint.net/` for more information.

Within the interface container we have the name of the fictional company, some basic instructions for using the page, a list of the different charges, and an `<input>` element to allow weights to be entered.

Most of the CSS that we added in this task was purely decorative and specific to this example. It could easily be wildly different if a different look and feel was required. We've made the map container take up the full width and height of the page, and styled the interface so that it appears to float over the right-hand side of the page.

Initializing the map

Getting a zoomable and panable interactive Google map up and running takes a ludicrously small amount of code. In this task we'll add that code, as well as set up some of the variables that we'll use later in the script.

Prepare for Lift Off

In this task we'll initialize the variables needed to configure the map and make a call to the Google Maps API. We should start by adding the standard jQuery wrapper to the empty `google-map.js` file that we created earlier:

```
$(function () {
    //all other code in here...
});
```

Remember, the `$(function () { … });` construct is a shortcut for jQuery's `document.ready` event handler.

Engage Thrusters

Within the wrapper we just added, we should add the following code:

```
var api = google.maps,
    mapCenter = new api.LatLng(50.91710, -1.40419),
    mapOptions = {
        zoom: 13,
        center: mapCenter,
        mapTypeId: api.MapTypeId.ROADMAP,
        disableDefaultUI: true
    },
    map = new api.Map(document.getElementById("map"), mapOptions),
    ui = $("#ui"),
    clicks = 0,
    positions = [];
```

Objective Complete - Mini Debriefing

In this task we start by creating some variables that we'll need to initialize the map. We'll be addressing the `google.maps` namespace throughout our code so the first variable we set is the contents of the top two namespaces for convenience.

Having a locally scoped copy that reaches right into the actual API that we want to use will make our code marginally more efficient because it is easier for our code to resolve a single variable. It's also much quicker to type in the first place.

All properties and methods used by the Google Maps API are namespaced. They all sit within the `maps` namespace, which itself sits in the `google` namespace. Google has such a large code-base for use in so many different applications that it makes sense to keep everything isolated and organized using namespaces.

For an excellent in-depth discussion on the intricacies of namespacing in JavaScript, see the excellent article on the subject by JavaScript supremo *Addy Osmani* (`http://addyosmani.com/blog/essential-js-namespacing/`).

Next we store the latitude and longitude that we'd like to center the map on. This is done using the Google Maps API's `LatLng()` method, which takes two arguments, the latitude and longitude values, and returns a `LatLng` object for use with other API methods. Notice how we call the `LatLng` constructor using our local `api` variable.

We can then create an object literal containing some of the configuration options that our map will need. These options include the zoom level, the location the map should be centered on, the type of map, and an option which disables the default map type and zoom/pan controls. We can use the `LatLng` object contained in `mapCenter` for the `center` configuration option.

Following this we create a new map instance using the map API's `Map()` constructor function. This function accepts two arguments: the first is the DOM element that the map should be rendered into and the second is the object literal containing the configuration options that we wish to set.

The first argument takes an actual DOM element, not a jQuery-wrapped DOM element. So although we could select the element from the page using jQuery and then extract the raw DOM element, it is more efficient to use JavaScript's native `getElementById()` function to retrieve the map container we added to the page in the previous task and pass it to the `Map()` constructor.

Next, we cache a jQuery selector for the UI container so that we can access it from the page repeatedly without having to actually select it from the DOM each time, and define a variable called `clicks`, which we'll use to record how many times the map has been clicked. We need to define it here in the top-level function scope so that we can reference it from within a click handler later in the code.

Lastly, we add an empty array literal in the variable `positions`, which we'll populate later on when we need to store the different areas of the map that have been clicked on. The array needs to be in the scope of the top-level function so that we can access it from within different event handlers later in the code.

Showing the company HQ with a custom overlay

In this task we'll put the company HQ on the map, literally, by adding a custom marker complete with an overlay that provides some basic information about the company, and perhaps an image of the premises.

Prepare for Lift Off

In this task we'll cover the following subtasks:

- Adding a marker to the map
- Adding a hidden element containing information about the company
- Adding a custom overlay to display the company information when the new marker is clicked
- Adding a click handler to show the overlay when the marker is clicked

Engage Thrusters

Adding a custom marker to the map can be achieved with the following simple code block, which should be added directly after the variables we added in the previous task:

```
var homeMarker = new api.Marker({
    position: mapCenter,
    map: map,
    icon: "img/hq.png"
});
```

To create an information overlay, or info window to use the correct Google terminology, for our new marker, we should first add an HTML element that contains the content we wish to display in the overlay. We can add the following new collection of elements to `google-map.html` directly after the UI container:

```
<div id="hqInfo">
    <img class="float-left" src="http://placehold.it/140x100"/>
    <h1>I Am Mover</h1>
    <p>This is where we are based.</p>
    <p>Call: 0123456789</p>
    <p>Email: info@i-am-mover.com</p>
</div>
```

We're using the `placehold.it` service again so that we don't have to worry about sourcing or creating an actual image for this bit of example content. It's a great service to use when mocking up prototypes quickly.

To tell the map about the new info window, we can use the following code, which should be added directly after the `homeMarker` code back in `google-map.js`:

```
var infoWindow = new api.InfoWindow({
    content: document.getElementById("hqInfo")
});
```

We also need some extra CSS to style the contents of the info window and to hide it until it is required. Add the following code to the bottom of `google-map.css`:

```
body > #hqInfo { display:none; }
#hqInfo { width:370px; }
#hqInfo h1 { margin-bottom:.25em; line-height:.9em; }
#hqInfo p { margin-bottom:.25em; }
```

Finally, we can add a simple click handler that displays the info window using the following code, which should be added after the `infoWindow` variable that we added a moment ago in `google-map.js`:

```
api.event.addListener(homeMarker, "click", function(){
    infoWindow.open(map, homeMarker);
});
```

Objective Complete - Mini Debriefing

First of all we defined a new marker, which is done using Google's `Marker()` constructor. This function takes a single argument, which is an object literal that defines different properties of the marker.

We set the `position` of the marker to be the center of the map for simplicity, although when defining other markers you'll see that any `LatLng` object can be used. We should also define the map that the marker belongs to, which we set to the `map` variable that contains our map instance. To specify the image to use as the marker, we can supply a relative path in string format to the `icon` option.

We then added a new container to the page which contains the information we want to display in our custom info window. The content here is not important; it's the technique that matters. We also added some additional styling for the contents of the info window.

In order to add the info window to our map instance, we used Google's `InfoWindow()` constructor. This method also takes a single argument, which again is an object literal which contains the options we wish to set. In this example we just set the `content` option to the element containing the content we just added to the page.

This should be an actual DOM element, hence we use JavaScript's `document.getElementById()` to get the element, instead of selecting it with jQuery.

Lastly we added an event handler to the map using Google's `addListener()` method. This method takes the element to attach the event handler to, which in this case is the marker we added, as the first argument, the event we wish to listen for as the second argument, and the callback function to handle the event as the third argument. The signature of this method is very similar to the event handling methods found in other common JavaScript libraries, although it is slightly different to how events handlers are added in jQuery.

Within the anonymous function we pass as the third argument to the `addListener()` method, all we do is call the `open()` method of our info window. The `open()` method takes two arguments; the first is the map that the info window belongs to, and the second is the location the info window is added to, which we set to our marker.

At this point we should be able to run the page in a browser, click on our custom marker, and have the contents of our hidden `<div>` displayed in the info window, as shown in the following screenshot:

Capturing clicks on the map

In this task we need to add a click handler for our map so that visitors can set the start and end of their transportation journey.

Engage Thrusters

First of all we need to add the function that will be executed when the map is clicked. Directly after the listener that we added in the last task, add the following function expression:

```
var addMarker = function (e) {

    if (clicks <= 1) {

        positions.push(e.latLng);

        var marker = new api.Marker({
            map: map,
            position: e.latLng,
            flat: (clicks === 0) ? true : false,
            animation: api.Animation.DROP,
            title: (clicks === 0) ? "Start" : "End",
            icon: (clicks === 0) ? "img/start.png" : "",
            draggable: true,
            id: (clicks === 0) ? "Start" : "End"
        });

        api.event.trigger(map, "locationAdd", e);

    } else {
        api.event.removeListener(mapClick);
        return false;
    }
}
```

Then, to attach a listener for clicks on the map which fires this function, we can add the following code directly after it:

```
var mapClick = api.event.addListener(map, "click", addMarker);
```

Objective Complete - Mini Debriefing

First of all we added the function that will be executed every time the map is clicked. The function will automatically be passed the event object by the `addListener()` method, which will contain a `latLng` object for the coordinates on the map that were clicked.

The first thing we do in the function is store the `latLng` property of the event object in our `positions` array. We'll need to know both of the locations that were clicked so it is useful to add them both to the `positions` array, which is visible throughout our code.

Then we check whether the `clicks` variable that we defined earlier is less than or equal to `1`. Provided it is, we go ahead and create a new marker using Google's `Marker()` constructor. We used the constructor earlier when we added a marker to show the company's headquarters, but this time we set some different properties.

We set the `map` property to be our map instance, and this time set the `position` of the marker to the `latLng` object contained in the event object, which will match the point on the map that was clicked.

We'll use a green marker image for the first click, which will represent the start of the journey. The image we'll use already has its own shadow, so when we add the first marker, which we can determine using a JavaScript ternary that checks whether `clicks` is equal to `0`, we set the `flat` property to `true` to disable the shadow that Google will otherwise add.

We can easily add a nice drop animation so that when the map is clicked, the new marker drops into place. The animation features a bounce easing effect, which is also visually pleasing. The animation is set using the `animation` property, which is set to `DROP` using the `Animation` API.

We can also set a `title` for the marker, which is displayed when the cursor hovers over it, using the `title` property. Again we use a simple JavaScript ternary to set either the `Start` or `End` as the string depending on value of our `clicks` variable.

We use the `icon` property to specify the path to the image that we'll use for the start marker. When `clicks` is not equal to `0` we just specify an empty string, which causes the default red marker to be added.

We also set the `draggable` property to `true` to make the markers draggable. This will let users modify the start or end locations of the journey if they wish. We can add the code that will handle this a little later on.

Next we can use Google's `event` API to trigger a custom event. We use the `trigger()` method, specifying the `map` instance as the object that the event will originate from, `locationAdd` as the name of our custom event, and pass the event object that we've worked with in our `addMarker()` function (stored in `e`) as a parameter to any handlers that may be listening for our custom event. We add a handler for this event in the next section.

Lastly we can set a unique `id` attribute on the marker so that we can differentiate each marker. We'll need this when we want to update our UI following a marker drag, which we'll look at a little later on.

This is everything we want to do at this point while the `clicks` variable is still less than or equal to `1`. The second branch of the outer conditional in our `addMarker()` function deals with situations when `clicks` is greater than `1`.

In this case, we know the map has already been clicked twice, so when this occurs we want to stop listening for clicks on the map. We can unbind our handler using the `event` API's `removeListener()` method. This method simply takes a reference to the `eventListener` returned by the `addListener()` method.

When we bind the click event on the map to our `addMarker` function, we store what is returned in the `mapClick` variable, which is what is passed to the `removeListener()` method.

At this point we should be able to run the page in a browser and add new markers to the map by clicking at different locations.

Classified Intel

We used a **function expression** in this task, by assigning the event handler to a variable, instead of perhaps the more familiar **function declaration**. This is generally considered a good practice, and while not essential in this situation, it is certainly a good habit to get into. For a thorough understanding of why function expressions are generally better than function declarations, see *John Resig's* blog post at `http://ejohn.org/blog/javascript-as-a-first-language/`.

Updating the UI with the start and end locations

Once the two markers have been added to the map, we want to display their locations in the UI sidebar at the right of the page ready for when we compute the cost of the journey.

We'll want to show the full street address of each location that is clicked and also add a button that triggers the computation of a quote based on the locations that the visitor has chosen on the map.

Prepare for Lift Off

In the last task we used Google's `trigger()` method to trigger a custom event each time a new marker was added to the map following a click. In this task we'll add a handler for that custom event.

So far in this project, we've stuck almost entirely to Google's map API and haven't really used jQuery at all other than to add the initial `document.load` wrapper for the rest of code. In this part of the project we'll rectify that and fire up jQuery in order to update our UI.

Engage Thrusters

The handler for our custom `locationAdd` event should be as follows, which can be added directly after the `mapClick` variable from the last task:

```
api.event.addListener(map, "locationAdd", function (e) {

    var journeyEl = $("#journey"),
        outer = (journeyEl.length) ? journeyEl : $("<div>", {
            id: "journey"
        });

    new api.Geocoder().geocode({
        "latLng": e.latLng },
        function (results) {

            $("<h3 />", {
                text: (clicks === 0) ? "Start:" : "End:"
            }).appendTo(outer);
            $("<p />", {
                text: results[0].formatted_address,
                id: (clicks === 0) ? "StartPoint" : "EndPoint",
                "data-latLng": e.latLng
```

```
            }).appendTo(outer);

            if (!journeyEl.length) {
                outer.appendTo(ui);
            } else {
                $("<button />", {
                    id: "getQuote",
                    text: "Get quote"
                }).prop("disabled", true).appendTo(journeyEl);
            }

            clicks++;
        });
});
```

As we'll be adding some new elements to the page, we'll also need to update our style sheet for this project. Add the following new styles to the bottom of `google-map.css`:

```
#journey { margin-top:2em; }
#journey h3 { margin-bottom:.25em; }
```

Objective Complete - Mini Debriefing

We add the event handler for our custom `locationAdd` event in the same way that we added our click events, using Google's `addListener()` method.

Within the event handler we first define some variables. The first is a cached jQuery object that represents the element that displays the start and end points.

The next variable we set is then one of two things. If the jQuery object we set as the first variable has length, we know the journey element exists on the page, so we just store a reference to it. If it doesn't exist, we create a new element to use as the journey element and set its `id` to `journey`.

When the map is clicked for the first time, the journey element won't exist and will be created. The second time the map is clicked, the element will exist, so it will be selected from the page instead of being created.

Next we use the `geocode()` method of Google's `Geocoder()` API, which allows us to reverse-geocode a `latLng` object to get a street address. This method takes two arguments. The first is a configuration object, which we can use to specify the `latLng` object that we want to convert.

The second argument is a callback function that is executed once the geocoding is complete. This function is automatically passed a `results` object that contains the address.

Within this callback function we can use jQuery to create new elements to display the address and then append them to the journey element. The complete street address is found in the `formatted_address` property of the `results` object, which we can set as the text of one of the new elements. We can also set an `id` attribute on this element so that we can easily select it programmatically when required, and store the `latLng` object of the location using a custom `data-latLng` attribute.

The `results` object also contains a range of other useful properties about the address, so be sure to check it out in the object explorer of your favorite browser-based developer toolkit.

If the journey element doesn't exist we can then append it to the UI in order to display the address of the location. If it does exist, we know that it is the second click and can then create a new `<button>` that can be used to generate a quote based on the distance between the two locations.

We disable the `<button>` element using jQuery's `prop()` method to set the `disabled` property. We can enable the button later when a weight is added to the `<input>` in the UI.

Once we have added the new elements showing the journey start and end points in the UI, we can then increment the `clicks` variable so that we can keep track of how many markers have been added.

Now when we run the page and click on the map twice to add both the markers, the address of the points that we clicked should be displayed in the UI area at the right of the page. We should also now see the red end marker and be limited to adding only two markers now that we're incrementing the `clicks` variable.

Handling marker repositions

We've made our map markers draggable, so we need to handle address changes following a marker drag. This task will show just how easily that can be done. This will take just two steps:

- Binding each marker to the `dragend` event
- Adding the handler function for the event

Engage Thrusters

First we need to bind each marker to the `dragend` event when the marker is created. To do this, we should add the following highlighted line of code to the `addMarker()` function, directly after the marker's constructor:

```
var marker = new api.Marker({
    map: map,
    position: e.latLng,
```

```
        flat: (clicks === 0) ? true : false,
        animation: api.Animation.DROP,
        title: (clicks === 0) ? "Start" : "End",
        icon: (clicks === 0) ? "img/start.png" : "",
        draggable: true,
        id: (clicks === 0) ? "start" : "end"
    });

    api.event.addListener(marker, "dragend", markerDrag);
```

Next we should add the `markerDrag()` function itself. This can go directly after the `locationAdd` handler that we added in the last task:

```
var markerDrag = function (e) {
    var elId = ["#", this.get("id"), "Point"].join("");

    new api.Geocoder().geocode({
        "latLng": e.latLng
    }, function (results) {
        $(elId).text(results[0].formatted_address);
    });
};
```

Objective Complete - Mini Debriefing

In this task we first updated the `addMarker()` function to bind each new marker to the `dragend` event, which will be fired once the marker stops being dragged. We specify the marker as the first argument to Google's `addListener()` method, which is the object to bind to the event. The name of the event, `dragend`, is specified as the second argument, and `markerDrag` as the name of the function that will handle the event.

Then we added `markerDrag()` as a function expression. Because it's an event handler it will automatically be passed to the event object, which once again contains the `latLng` that we need to pass to a `Geocoder()` to get the new address that the marker was dragged to.

Inside the handler we first set a new variable that will be used as the selector for the element in the UI we want to update. Instead of concatenating a string together, we use the `array.join()` technique for performance reasons. The first and last items in the array we join are simply text.

The second item will be a string containing either `start` or `end` depending on which marker was dragged. Inside our event handler this refers to the marker, so we can use it get the custom `id` property that we added to each marker when it was created, allowing us to update the right element in the UI.

Once we have constructed the selector we just get the street address using Google's `geocode()` method exactly as we did before, which will give us the new address of the marker after the drag.

Inside the callback function for `geocode()` we use the selector we just created to select the `<p>` element in the UI and update its text content to the newly geocoded address.

Now when we view the page, we should be able to add the markers to the map as before, then drag them around and see the new address in the UI area at the right of the page.

Factoring in weights

We now have two addresses – the start and end markers for the journey. All the visitor needs to do now is enter a weight and we'll be able to calculate and display the distance and cost.

Engage Thrusters

All we need to do in this task is add a handler for the `<input>` element in the UI area so that once a weight is entered the `<button>` becomes clickable. We can achieve this with the following code, which can be added directly after the `markerDrag()` function from the previous task:

```
$("#weight").on("keyup", function () {
    if (timeout) {
        clearTimeout(timeout);
    }

    var field = $(this),
        enableButton = function () {
            if (field.val()) {
                $("#getQuote").removeProp("disabled");
            } else {
                $("#getQuote").prop("disabled", true);
            }
        },
        timeout = setTimeout(enableButton, 250);
});
```

Objective Complete - Mini Debriefing

We can add the event handler for the user-generated `keyup` DOM event using jQuery's `on()` method. Using the `on()` method is now the standard way of attaching event handlers in jQuery. Old methods such as `live()` or `delegate()` have now been deprecated and should not be used.

Within the event handler we first check whether a timeout has been set, and if it has, we clear it.

We then cache a selector for the `<input>` element so that we can see it inside our `enableButton()` function. We add the `enableButton()` function, again as a function expression.

All this function does is check whether the `<input>` element has a value, and if it does, we set the `disabled` property to `false` using jQuery's `prop()` method. If it doesn't have a value, we just disable it once more by setting the `disabled` property to `true`. Lastly we set a timeout using the JavaScript `setTimeout()` function, passing it the `enableButton()` function as the first argument. We set `250`, or a quarter of a second, as the timeout length. The timeout is stored in the `timeout` variable, ready for us to check the next time the function is executed.

Classified Intel

The reason we use the timeout here is to rate-limit the number of times the `enableButton()` function is executed. The function will be invoked after every character is entered into the field.

A quarter of a second is a barely discernible delay, but if someone types a long number into the field quickly, it can drastically reduce the number of times the function runs. Within the function, we select an element from the page and create a jQuery object. That's not too intense and in this example we probably don't even need to worry about it. But using a timeout like this is a robust solution that can help out when doing more intense operations inside a frequently fired event handler.

We could have just used jQuery's `one()` method to attach an event handler that simply enables the `<button>` and then removes itself. However, this wouldn't allow us to disable the `<button>` once more if the figure entered into the field is removed.

Displaying the projected distance and cost

Our last task in this project is to get the distance between the two markers and calculate the cost of the journey. Once calculated, we should probably display the results to the visitor too.

Engage Thrusters

First we should attach a click event handler for our `<button>`. Add the following code directly after the handler for the `keyup` event that we added in the last task:

```
$("body").on("click", "#getQuote", function (e) {
    e.preventDefault();

    $(this).remove();
});
```

Next we can get the distance between the two points. Directly after the `remove()` method we just added (but still inside the click handler function), add the following code:

```
new api.DistanceMatrixService().getDistanceMatrix({
    origins: [$("#StartPoint").attr("data-latLng")],
    destinations: [$("#EndPoint").attr("data-latLng")],
    travelMode: google.maps.TravelMode.DRIVING,
    unitSystem: google.maps.UnitSystem.IMPERIAL
}, function (response) {

});
```

Now we just need to compute and display the cost, which we can do by adding the following code to the empty callback function we just added. First we can add the variables we'll need:

```
var list = $("<dl/>", {
        "class": "clearfix",
        id: "quote"
    }),
    format = function (number) {
        var rounded = Math.round(number * 100) / 100,
            fixed = rounded.toFixed(2);

        return fixed;
    },
    term = $("<dt/>"),
    desc = $("<dd/>"),
```

```
    distance = response.rows[0].elements[0].distance,
    weight = $("#weight").val(),
    distanceString = distance.text + "les",
    distanceNum = parseFloat(distance.text.split(" ")[0]),
    distanceCost = format(distanceNum * 3),
    weightCost = format(distanceNum * 0.25 * distanceNum),
    totalCost = format(+distanceCost + +weightCost);
```

Next we can generate the HTML structure that we'll use to display the computed figures:

```
$("<h3>", {
    text: "Your quote",
    id: "quoteHeading"
}).appendTo(ui);

term.clone().html("Distance:").appendTo(list);
desc.clone().text(distanceString).appendTo(list);
term.clone().text("Distance cost:").appendTo(list);
desc.clone().text("£" + distanceCost).appendTo(list);
term.clone().text("Weight cost:")
            .appendTo(list);

desc.clone().text("£" + weightCost).appendTo(list);
term.clone().addClass("total").text("Total:").appendTo(list);
desc.clone().addClass("total")
            .text("£" + totalCost)
            .appendTo(list);

list.appendTo(ui);
```

Lastly, we should probably add some additional styling for the new elements that we just created and added to the page. At the bottom of `google-map.css`, add the following new styles:

```
#quoteHeading {
    padding-top:1em; border-top:1px dashed #aaa;
    margin-top:1em;
}
#quote dt { margin-right:0; }
#quote dd { width:50%; }
#quote .total {
    padding-top:.5em; border-top:1px dashed #aaa;
    margin-bottom:0; font-size:1.5em;
}
```

Objective Complete - Mini Debriefing

We started out by binding a click event handler to the `body` of the page using jQuery's `on()` method. This time we use the 3-argument form of the method where the first argument is still the name of the event, the second argument is a selector to filter the event by, and the third argument is the function to trigger when the event occurs.

Events in JavaScript bubble up through their containers and when the event hits the `body`, it will be filtered by the selector used as the second argument and the function will only be executed if it was dispatched by an element that matches the selector. In this example, only events dispatched by the `<button>` will trigger the function.

Using the `on()` method in this form gives us a means of employing powerful event delegation that allows us to bind events for elements which may or may not exist at the time of the binding.

Within the handler function, we first prevent the default behavior of the browser. There shouldn't be any default behavior because we don't have a `<form>` on the page so there is nothing for the `<button>` to submit. But if someone were to try and run this on an ASPX page, which usually does have a `<form>` enclosing most, if not all, of the elements on the page, it could behave in unexpected ways. Unless strictly necessary, `preventDefault()` should always be used.

We then remove the `<button>` from the page. Note that even though the event handler is bound to the `<body>`, the `this` object inside the handler function still points at the `<button>` element that triggered the event.

We then used another of Google's APIs – the `DistanceMatrixService()`, which allows us to compute the distance between two or more points on the map. Because we don't need to reference the object returned by the `DistanceMatrixService()` constructor, we can chain the `getDistanceMatrix()` method directly onto it.

This method takes two arguments with the first being a configuration object and the second a callback function to execute when the method returns. The callback function will automatically be passed an object containing the response.

We set several configuration options using the first argument. The `origins` and `destinations` options both take arrays where each item in each array is a `latLng` object. We can easily get the `latLng` objects for both of the markers using the custom `data-latLng` attribute that we set when we showed the addresses.

We also set the `travelMode` option to the distance it would be via road using the `google.maps.TravelMode.DRIVING` constant, and set the `unitSystem` option to `google.maps.UnitSystem.IMPERIAL` to give a distance in miles instead of kilometers, for no other reason than because I'm a Brit, and I'm used to using miles.

The callback function we supply is automatically passed a results object that contains, of course, the results returned by the distance matrix. The first half of the callback function is concerned with creating variables and computing values. The second part of the function deals with displaying the information that has been computed.

We first create a new `<dl>` element and give it a `class` that is required for use with our `common.css` style sheet, and an `id` attribute, mostly for decorative styling. Then we add a simple function expression that receives a number as an argument, rounds it, and then fixes it to two decimal places before returning it. We'll use this function to ensure that our financial figures are in the required format.

We also create a new `<dt>` element and a new `<dd>` element that we can clone as many times as required without having to repeatedly create new instances of jQuery, and then store the value entered into the weight text field using jQuery's `val()` method.

Next we extract the `distance` property from the object passed to the callback function. Its structure may look complex, as the object we are actually interested in for this example is buried within a multidimensional array, but as the method's name suggests, it is able to return a complex matrix of results for multiple origins and destinations.

Following this we concatenate a string that includes the `text` property of the `distance` object that we just stored and the full word `miles`. The distance matrix returns imperial results as `mi` instead of the full `miles`, so we add the string `les` to the end of the value.

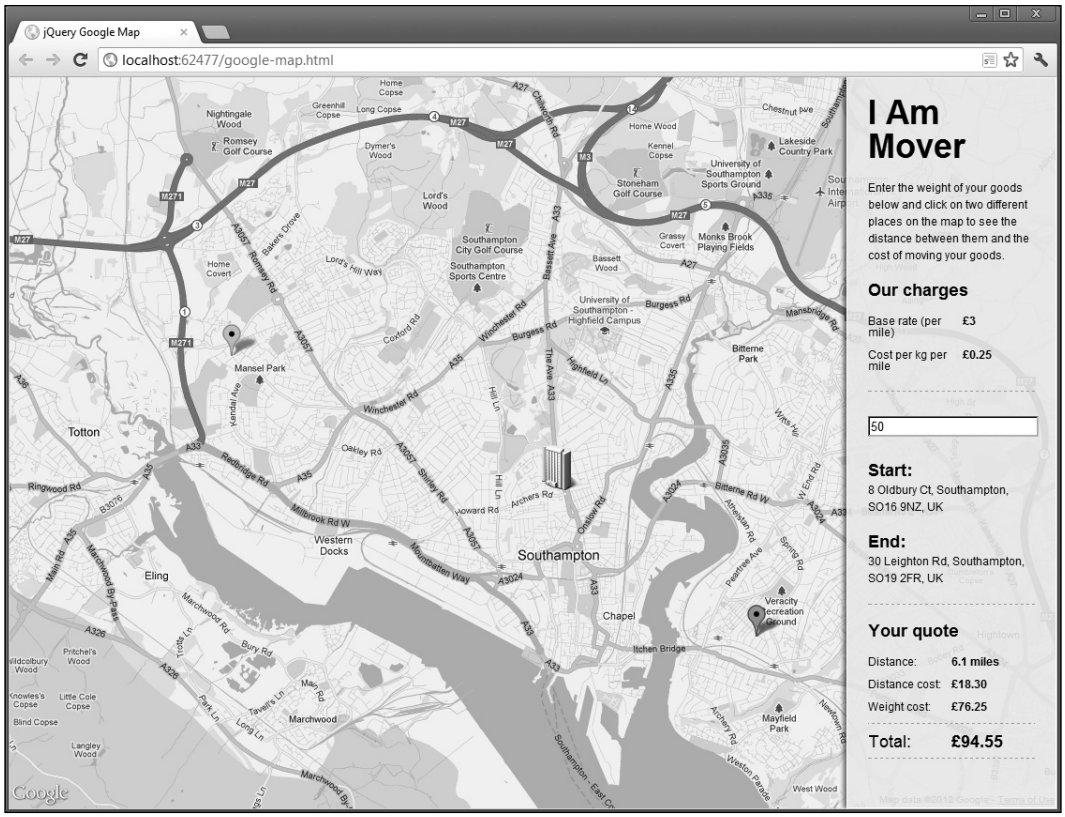
We then get the numerical distance by splitting the string on the space between the number of miles and the letters `mi`. JavaScript's `split()` function will return an array of two items containing the part of the string up to, but not including, the split-character and the part after the split-character. We are only interested in the first item in this array, and also use JavaScript's `parseFloat()` function to ensure that this value is definitely a number and not a string.

Now we have enough information to actually work out the cost of the journey. We've specified the charge per mile to be £3 so we multiply the distance by `3` and pass the result to our `format()` function so that the number is in the correct format.

We can also work out the charge per kilogram per mile in a very similar way, first multiplying the weight by the cost per kilogram, then multiplying by the distance. Again we pass this figure into our `format()` function. Then we can work out the total cost by adding these two figures together. The figures that we've been working with somehow become strings. To fix this, we can still use our `format()` function, but we prefix each of the values we want to add with the `+` character, which will force them to be numbers and not strings.

Once we have created the figures we wish to display, we can then create the new elements that we need to use to display them, starting with a nice heading to help clarify the new set of information we're adding to the UI.

We can then create the clones of the `<dt>` and `<dd>` elements which hold each label and figure. Once these have been created, we append them to the `<dl>` element we created, before finally appending the new list as a whole to the UI, as shown in the following screenshot:



Classified Intel

The astute of you will notice that the number rounding solution we've used in this example isn't that robust, and won't round all fractions as precisely (or correctly) as would be required for a genuine system that deals with real currency.

JavaScript does not handle floating point arithmetic as gracefully as some other languages do, and so creating the perfect rounding system that rounds correctly 100 percent of the time is beyond the scope of this book.

For those who are interested, the stackoverflow site has some extremely illuminating answers posted to questions around currency formatting in JavaScript. For example, see: `http://stackoverflow.com/questions/149055/how-can-i-format-numbers-as-money-in-javascript`.

Mission Accomplished

We've covered a lot of both Google and jQuery functionality in this project. Specifically we looked at the following subjects:

- Adding markers and overlays to the map using the `Marker()` and `InfoWindow()` constructors.
- Reacting to map-driven events such as clicks on markers or marker drags. Event handlers are attached using the `addListener()` method of the `google.maps` API. We also saw how to fire custom events using the `trigger()` method.
- Using Google's services to manipulate the data generated by the map. The services we used were the `Geocoder()` to reverse-geocode the `latLng` of each point on the map that was clicked in order to obtain its address, and the `DistanceMatrixService()` to determine the distance between the points.
- Taking advantage of jQuery's event capabilities to add both standard and delegated events using the `on()` method to detect when different parts of our UI were interacted with, such as the `<button>` being clicked or the `<input>` being typed into.
- Using jQuery's powerful DOM manipulation methods to update the UI with addresses and the quote. We used a range of these methods including `clone()`, `html()`, `text()`, and `prop()`, as well both selecting and creating new elements.

You Ready To Go Gung HO? A Hotshot Challenge

In this example, visitors are only able to generate a single quote. Once the `getQuote` `<button>` is clicked, the results are displayed and no further interaction is possible. Why don't you change it so that a reset button is added to the UI when the quote is generated? The visitor can then clear the quote and the markers from the map and start over from scratch.

Where to buy this book

You can buy jQuery HOTSHOT from the Packt Publishing website:
`http://www.packtpub.com/jquery-hotshot/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/jquery-hotshot/book