2
jQuery—the Most Common JavaScript Framework

JavaScript frameworks can offer at least two kinds of advantages, as we have discussed earlier. Firstly, they can offer considerable facilities when compared to building from scratch. TurboGears and Django are both server-side web frameworks in Python and both offer major advantages compared to web application development in Python using only the standard library, even with the CGI module included. Secondly, they can offer a more uniform virtual programming interface compared to writing unadorned JavaScript with enough detection and conditional logic to directly conform to disparate JavaScript environments so that code written in one browser and debugged in another browser stands a fighting chance of working without crashing in every common browser.

As regards the question, "Which is best?" if you only learn one client-side JavaScript framework, jQuery is an excellent choice. It is one of the most popular JavaScript frameworks, partly because it is (like Python) gentle to newcomers. However, comparing jQuery to Dojo is like comparing a hammer to a wrench. (A general JavaScript framework comparison is at `http://en.wikipedia.org/wiki/Comparison_of_JavaScript_frameworks`.) Unlike server-side web development frameworks such as Django and TurboGears, you don't necessarily ask, "Which one will we use for this project?" as there might be good reasons to use more than one framework in a web application. jQuery's developers know it isn't the only thing out there. It only uses two names in the global namespace: `$` and `jQuery`. Both do the same thing; you can always use `jQuery()` in lieu of `$()`, and if you call `jQuery.noconflict()`, jQuery will let go of `$` so that other libraries like Prototype (`http://www.prototypejs.org/`) are free to use that name.

Finally, we suggest that jQuery might be something like a "virtual higher-level language" built on top of JavaScript. A programmer's first reaction after viewing the JavaScript code, on seeing code using jQuery, might well be, "Is that really JavaScript? *It doesn't look like JavaScript!*" A programmer using jQuery can operate on element sets, and refine or expand, without ever typing out a JavaScript variable assignment or keyword such as `if`, `else`, or `for`, and even without cross-browser concerns, one runs into jQuery code that does the work of dozens of lines of "bare JavaScript" in a single line. But these are single lines that can be very easily read once you know the ground rules, not the cryptic one-liners of the Perl art form. This "virtual higher-level language" of jQuery is not Python and does not look like Python, but there is something "Pythonic in spirit" about how it works.

In this chapter, we will cover:

- How jQuery simplifies "Hello, world!" in Ajax
- The basic Ajax facilities jQuery offers
- How jQuery effectively offers a "virtual higher-level language"
- jQuery selectors, as they illustrate the kind of more Pythonic "virtual higher-level language facilities" that jQuery offers
- A sample "kickstart Django Ajax application," which covers every major basic feature except for server-side persistence management
- An overview of a more serious, in-depth Django application, to be covered in upcoming chapters

Let's explore the higher-level way to do things in jQuery.

jQuery and basic Ajax

Let's look at a "Hello, world!" in Ajax. A request containing one variable, `text`, is sent to the server, and the server responds with "Hello, [*contents of the variable text*]!" which is then put into the `innerHTML` of a paragraph with ID `result`:

```
if (typeof XMLHttpRequest == "undefined")
    {
    XMLHttpRequest = function()
        {
        try
            {
            return new ActiveXObject("Msxml2.XMLHTTP.6.0");
            }
        catch(exception)
            {
            try
                {
                return new ActiveXObject("Msxml2.XMLHTTP.3.0");
                }
            catch(exception)
                {
                try
                    {
                    return new ActiveXObject("Msxml2.XMLHTTP");
                    }
                catch(exception)
                    {
                    throw new Error("Could not construct
                        XMLHttpRequest");
                    }
                }
            }
        }
    }

var xhr = new XMLHttpRequest();
xhr.open("GET", "/project/server.cgi?text=world");
callback = function()
    {
    if (xhr.readyState == 4 && xhr.status >= 200 && xhr.status < 300)
        {
        document.getElementById("result").innerHTML =
          xhr.responseText;
        }
    }
xhr.onreadystatechange = callback;
xhr.send(null);
```

With the simpler, "virtual high-level language" of jQuery, you accomplish more or less the same with only:

```
    $("#result").load("/project/server.cgi", "text=world");
```

`$()` selects a wrapped set. This can be a CSS designation, a single HTML entity referenced by ID as we do here, or some other things. jQuery works on wrapped sets. Here, as is common in jQuery, we create a wrapped set and call a function on it. `$.load()` loads from a server URL, and we can optionally specify a query string.

How do we do things with jQuery? To start off, we create a **wrapped set**. The syntax is based on CSS and CSS2 selectors, with some useful extensions.

Here are some examples of creating a wrapped set:

```
$("#result");
$("p");
$("p.summary");
$("a");
$("p a");
$("p > a");
$("li > p");
$("p.summary > a");
$("table.striped tr:even");
```

They select nodes from the DOM more or less as one would expect:

- `$("#result#")` selects the item with HTML ID `result`
- `$("p")` selects all paragraph elements
- `$("p.summary")` selects all paragraph elements with class `summary`
- `$("a")` selects all anchor tags
- `$("p a")` selects all anchor tags in a paragraph
- `$("p > a")` selects all anchor tags whose immediate parent is a paragraph tag
- `$("li > p")` selects all `p` tags whose immediate parent is a `li` tag
- `$("p.summary > a")` selects all anchor tags whose immediate parent is a paragraph tag with class `summary`
- `$("table.striped tr:even")` selects even-numbered rows from all tables belonging to class `striped`

The difference between `$("p a")` and `$("p > a")` is that the first selects any tag contained in a `p` tag, whether its immediate parent is the `p` tag or whether there are intervening span, em, or strong tags, and the second selects only those tags whose immediate parent is a `p` tag; the latter yields a subset of the former. This last approach lends itself to a straightforward and clean way to address the chore of tiger-striping tables:

```
$("table.striped tr:even").addClass("even");
```

The statements before this last example are not useful in themselves, but they lay a powerful foundation by creating a wrapped set. A wrapped set is an object that encapsulates both a set of DOM elements and a full complement of operations, such as `.addClass()`, which assigns a CSS class to all elements from a wrapped set, or `.load()`, which provided a one-line Ajax solution earlier.

One of the key features of a wrapped set is that these operations return the same wrapped set, unless they are one of a few exceptions, such as operations designed to change the set by adding or removing members. This means that operations can be **chained**; for a somewhat artificial example, our code to stripe items in a table could be extended to slowly hiding them, waiting for a second (specified as 1000 milliseconds), and then showing them again:

```
$("table.striped tr:even").addClass("even").hide("slow").delay(1000).
show("slow");
```

In this statement, all of the operations return the same wrapped set. Dozens of further operations could be appended after the `.show()` call if desired. In chaining of operations, the additional functions can function as separate "virtual statements". The previous line of code, in pseudocode, could be written:

```
Select even-numbered table rows from tables having the class
"striped".
Add the class "even" to them.
Slowly hide them.
Wait 1000 milliseconds.
Slowly show them.
```

The final appearance of the page should be as if merely `$("table.striped tr:even").addClass("even");` had been called.

jQuery Ajax facilities

jQuery provides good facilities both for Ajax and other JavaScript usage. The API is excellent and can be bookmarked from `http://api.jquery.com/`.

We will be going through some key facilities, Ajax and otherwise, and making comments on how these facilities might best be used.

$.ajax()

The most foundational and low-level workhorse in jQuery is `$.ajax()`. It takes the arguments as shown in the following *Sample invocation*. Default values can be set with `$.ajaxSetup()`, as discussed below:

```
$.ajax({data: "surname=Smith&cartTotal=12.34", dataType: "text",
error: function(XMLHttpRequest, textStatus, errorThrown) {
        displayErrorMessage("An error has occurred: " + textStatus);
        }, success: function(data, textStatus, XMLHttpRequest) {
        try
            {
            updatePage(JSON.parse(data));
            }
        catch(error)
            {
            displayErrorMessage("There was an error updating your
               shopping cart. Please call customer service at
               800-555-1212.");
            }
        },  type: "POST", url: "/update-user"};
```

Some of the fields that can be passed are described in the following sections.

context

The context, available as the variable `this`, can be used to give access to variables in callbacks, even if they have fallen out of scope. The following code prompts for the user's name and e-mail address and provides them in a context, anonymously, so that the data are available to the callback function but are never a part of the global namespace.

```
    $.ajax({success: function(data, textStatus, XMLHttpRequest)
            {
            alert(this.name + ", your email address is " +
                this.email + ".");
            processData(data, this.name, this.email);
            }, context:
            {
            name: prompt("What is your name?", ""),
            email: prompt("What is your email address?", "")
            }, …
        });
```

Closures

The principle at play is the principle of **closures**. The wrong way, perhaps, to learn about closures is to look for academic-style introductions to what you need to know in order to understand them. There's a lot there, and it's a lot harder to understand than learning by jumping in. Closures represent a key concept in core JavaScript, along with other things such as functions, objects, and prototypes. Closures are part of the conceptual landscape and not only because they are the basis on how to effectively create an object with private fields. So we will jump in and give a closure that serves as a proof of concept as, effectively, an object with private fields. To give a standard, "bare JavaScript" example of using a closure to create an object with private variables, which creates an object that stores an integer value, has a getter and setter, but in Java fashion ensures that the field can only have an integer value.

We define a function, which we will be immediately evaluating; `closureExample` stores not the anonymous function but its return value. The variable field is a local variable:

```
var closure_example = function()
    {
    var field = 0;
    return {
```

The getter is an unadorned getter as in Java:

```
            get: function()
                {
                return field;
                },
```

The setter could be an unadorned setter, storing `newValue` in a field and doing nothing else. However, we provide a more discriminating behavior: we make a string of the object and then see if it can be parsed as an integer. This will be an integer if we obtain an integer and NaN (Not a Number) if it cannot be parsed as an integer. If the value is not a number, then we return false; if it gives us an integer, then we store the integer and return true:

```
             set: function(newValue)
                {
                var value = parseInt(newValue.toString());
                if (isNaN(value))
                    {
                    return false;
                    }
                else
                    {
                    field = value;
                    return true;
                    }
                }
            }
    } ();
```

This creates and evaluates an anonymous function. Local variables (in this case, `field`) do not enter the global namespace but also remain around, lurking, accessible to the object returned and its two members: functions that can still access `field`. As long as the object stored in `closure_example` is available, its members will have access to its local variables, which are not gone and garbage collected until the object in `closure_example` itself is gone and eligible for garbage collection.

We can use it as follows:

```
closure_example.set(3);
var retrieved = closure_example.get();
closure_example.set(1.2);
var assignment_was_successful = closure_example.set(1.2)
```

Prototypes and prototypal inheritance

Prototypes and **prototypal inheritance** are a basis for object-oriented programming, with inheritance, but without classes. In Java, certain features of a class are fixed. In Python, an object inherits from a class but features that are fixed in Java cannot be overridden. In JavaScript, we go one step further and say that objects inherit from other objects. Class-based objects run in a Platonic fashion, where there is an ideal type and concrete shadows or copies of that ideal type. Prototypal inheritance is more like the evolutionary picture of single-celled organisms that can mutate, reproduce asexually by cell division, and pass on (accumulated) mutations when they divide. In JavaScript, an object's prototype is set by, for instance:

```
customer.prototype = employee;
```

Redefine members for a customer when one wants to change from the attributes of an employee. (Members that are not redefined default to the prototype's values, and if not found on the prototype, default to members on the prototype's prototype, going all the way up the chain to the object if need be.) Also note that this inheritance may or may not mean moving from more general to more specific; inheritance may better be seen as "`customer` *mutates from* `employee`" than "`customer` is a *more specific type* of `employee`."

Let us return to the parameters of `$.ajax()`.

data

This is the form data to pass, whether given as a string as it would appear in a GET URI, as follows:

```
query=pizza&page=2
```

or given as a dictionary, as follows:

```
{page: 2, query: "pizza"}
```

dataFilter

This is a reference to a function that will be passed two arguments: the raw response given by an `XMLHttpRequest`, and a type (`xml`, `json`, `script`, or `html`). This offers an important security hook: JSON can be passed to an `eval()`, but malicious JavaScript that is passed in place of JSON data can also be passed to an `eval()`.

One of the cardinal rules of security on both the client-side and server-side is to treat all input as guilty until proven innocent of being malicious. Even though it is a "double work" chore, user input should both be validated at the client-side and the server-side: client-side as a courtesy to the user to improve the user experience and *not* as trustworthy security, and on the server side as a security measure against malicious data even if it is malformed malicious data that no normal web browser would send.

We might comment that the Django principle of **Don't Repeat Yourself** (**DRY**), is a major Django selling point but is *not* a reason to dodge handling both the user interface side of validation, and the security side. In practice, this means both validating from Django on the server-side and JavaScript on the client-side, for what we are doing. Needlessly repeating yourself in Django is a sign of bad code, but this is not *needless* repetition, even if it is a chore. It's *needed* repetition.

jQuery does not automatically include functionality to test whether something served up as JSON is malicious, and methods like `.getJSON()` trustingly execute what might contain malicious JavaScript. One serious alternative is the `JSON.parse()` method defined in `http://www.json.org/json2.js`. It will return a parsed object or throw a SyntaxError if it finds something suspicious.

dataType

This is something you should specify, and provide a default specified value via `$.ajaxSetup()`, for example:

```
$.ajaxSetup({dataType: "text"});
```

The possible values are `html`, `json`, `jsonp`, `script`, `text`, and `xml`. If you do *not* specify a value, jQuery will use an unsecure "intelligent guessing" that may trustingly execute any JavaScript or JSON it is passed without any attempt to determine if it is malicious. If you specify a `dataType` of `text`, your callback will be given the raw text which you can then test call `JSON.parse()` on. If you specify a `dataType` of `json`, you will get the parsed JSON object. So specify `text` even if you know you want JSON.

error(XMLHttpRequest, textStatus, errorThrown)

An error callback that takes up to three arguments. For example:

```
    $.ajax({error: function(XMLHttpRequest, textStatus, errorThrown)
            {
            registerError(textStatus);
            }, …
        });
```

success(data, textStatus, XMLHttpRequest)

A callback for success that also takes up to three arguments, but in different order. For example:

```
    $.ajax({success: function(data, textStatus, XMLHttpRequest) {
            processData(data);
            }, …
        });
```

If you want to do something immediately after the request has completed, the recommended best practice is to specify a callback function that will pick up after the request has completed. If you want to make certain variables available to the callback function that would not otherwise remain available, you may specify the context as discussed above.

type

The type of the request, for example GET, POST, and so on. The default is GET, but this is only appropriate in very limited cases and it may make sense to simply always use POST. The more serious the work you are doing, the more likely it is that you should *only* use POST. GET is appropriate only when it doesn't matter how many extra times a form is submitted. In other words, GET is only appropriate when there are no significant side effects, and in particular no destructive side effects. In a shopping cart application, GET may be appropriate to view the contents of a shopping cart, although POST is also appropriate and has the side effect of guaranteeing a fresh load. GET is not, however, appropriate for creating, modifying, fulfilling, or deleting an order, and you should use POST when any or all of these are involved.

url

The URL to submit to; this defaults to the current page.

$.aj0axSetup()

This takes the same arguments as `$.ajax()`. All arguments are optional, but you can use this to specify default values. In terms of DRY, if something can be appropriately offloaded to `$.ajaxSetup()`, it probably should be offloaded to `$.ajaxSetup()`.

Sample invocation

A sample invocation is as follows:

```
$.ajaxSetup({dataType: "text", type: "POST"});
```

$.get() and $.post()

These are convenience methods for `$.ajax()` that allow you to specify commonly used parameters without key-value hash syntax and are intended to simplify GET and POST operations. They both have the same signature.

The sample invocation is as follows:

```
$.get("/resources/update");
$.post("/resources/update");
$.post("/resources/update", "user=jsmith&product_id=112");
$.post("/resources/update", {user: "jsmith", product_id: 112});
$.post("/resources/update", function(data) { ("#result").html(data)
});
$.get("/resources/update", "user=jsmith&product_id=112",
  function(data, textStatus, XMLHttpRequest) {
        ("#result").html(data);
        logStatus(textStatus);
        });
$.post("/resources/update", "user=jsmith&product_id=112",
  function(data, textStatus, XMLHttpRequest) {
        ("#result").html(data);
        logStatus(textStatus);
        }, "text");
```

These are convenience methods for `$.ajax()` and make several common features from `$.ajax()` available, but notably not an error callback function. If you want a callback called in the case of an error for appropriate handling, as well as when everything goes perfectly well, register a global error handler or use `$.ajax()`.

For the real world, this is a substantial endorsement of not using the convenience methods alone, but either specifying a global error handler or using `$.ajax()`.

Something can go wrong in front of your boss, or worse, work perfectly when you show your boss and then proceed to blow up completely when your boss shows it to your customer. Heisenbugs, (subtle and hard-to-pin-down bugs that just show up under circumstances that are difficult to repeat), network errors, and server errors will occur, and unless a noop response is appropriate and production-ready behavior when an error prevents successful completion, you will want to specify an appropriate error callback.

Or, alternatively, it may be acceptable to specify a global error handler using `$.ajaxSetup({error: myErrorHandler})` if you can write something appropriate to generically but correctly handle all Ajax error conditions where you did not directly call `$.ajax()` and specify an error handler. This includes all calls to `$.get()`, `$.load()`, and `$.post()`.

.load()

This is a very convenient convenience method. If you're looking for a simple, higher-level alternative to `$.ajax()`, you can call this method on a wrapped set to load the results of an Ajax call into it. There are some caveats, however. Let's give some sample invocations, look at why `.load()` is attractive, and then give appropriate qualifications and warnings.

Sample invocations

The sample invocation is as follows:

```
$("#messages").load("/sitewide-messages");
$("#messages").load("/user-messages", "username=jsmith");
$("#hidden").load("/user-customizations", "username=jsmith",
function(responseText, textStatus, XMLHttpRequest) {
        performUserCustomizations(responseText);
        });
```

On the surface, this looks like a good example of an alternative to doing JavaScript work that incorporates jQuery, and instead using the "virtual higher-level language" that jQuery provides. `.load()`, like many jQuery functions, is a function of a wrapped set and returns a wrapped set (as often, the wrapped set it was given). It is, therefore, *in principle* something that can be put in a chain.

Why "in principle"? It makes sense, upon some condition, to hide all of the paragraphs in a form containing a checkbox that is not checked:

```
$("form p:has(input[type=checkbox]:not(:checked))").hide("slow");
```

This says, "For all `p` elements in a form that have an unchecked checkbox, slowly hide them."

Furthermore, one could want to do several actions instead of one:

```
$("form p:has(input[type=checkbox]:not(:checked))").
addClass("strikethru").delay(500).hide("slow");
```

That is a more elaborate animation: "For all `p` elements in a form that have an unchecked checkbox, add the class `strikethru` (which can be defined in CSS to have a strikethrough line through text), then wait half a second (500 milliseconds), and then slowly hide it as was done before."

But for a wrapped set like this, it would not make much sense to insert a `.load()` after the wrapped set is generated:

```
$("form p:has(input[type=checkbox]:not(:checked))").load("/updates");
```

What that says is, "For all `p` elements in a form that have an unchecked box, load the contents of the relative URL `/updates` and replace whatever the selected `p` elements contain with what was loaded." That's *legal*, but it's not as clear *why* someone would want to do this. Ordinarily, if you are going to load something from Ajax to add to the web page, it will make sense to insert it in one place and not every element in a multi-element wrapped set. So the possibility of calling `.load()` on any wrapped set, instead of a single DOM element as can be encapsulated in a wrapped set like `$.("#results")`, is not obviously such a terribly great improvement.

Furthermore, `.load()` will return immediately, not when things are loaded, so items following it in the chain should not be assumed to have the data loaded. But they also should not be assumed *not* to have the data loaded; we have a race condition, and we should only chain other items after `.load()` when race conditions about the order of execution do not matter. So the fact that we can chain other operations after `.load()`, which is arguably the glory of jQuery, does not mean that it is always wise to do so.

Another point to be made is as above: `.load()`, like `$.get()` and `$.post()`, effectively forces a noop error handler, and therefore should be used only when it is acceptable for nothing to be done when any of a number of things that can go wrong, do go wrong. There may be some cases where a noop is the best error handler, but usually best practices in Ajax are to give some feedback that something has gone wrong.

Convenience methods exist, but we recommend using `$.ajax()` because it provides callback facilities for error situations as well as for success, or writing an appropriate generic, all-purpose error callback to give to `$.ajaxSetup()` or equivalent (there are other alternatives, including `$.ajaxError()`). The best practices are to use a convenience method in conjunction with a global error handler, which can be made more sophisticated by examining the information in its arguments to tell what went wrong.

jQuery as a virtual higher-level language

There are some other basic functions that we have seen in jQuery besides direct Ajax. One example of other kinds of functions includes selectors.

The selectors

Selectors create a wrapped set. Some examples of selectors include:

- `$("*")`: Selects all DOM elements.
- `$(":animated")`: Selects all elements that are in the process of an animation at the time the selector is called.
- `$("id|=header")`: Selects all elements with attribute `id` (in this illustration), whose content matches `header` or `header-*`, like `header-image`. In this case, this would be an element ID. This and following selectors matching text are case sensitive, meaning that an ID of "HEADER" or even "Header" would not be matched.
- `$("value*=import")`: Selects all elements with attribute `value` containing the string `import`. This would include both strings like "Then import the following class." and "This is important."
- `$("value~=import")`: Matches elements having a value that contains the string `import`, but is delimited by spaces. This would include "Then we import the following class." but exclude "This is important."
- `$("id$=wrapper")`: Matches all elements having an ID that ends with `wrapper`. This would include an ID of "comment-wrapper" as well as `wrapper`.
- `$("http-equiv=refresh")`: Matches all elements where the `http-equiv` attribute exactly equals `refresh`.
- `$("id!=result")`: Matches all elements having IDs, where the ID does not equal `result`. This will not match elements that do not have IDs.
- `$("class^=main")`: Matches all elements having a class that begins with `main`.
- `$(":button")`: Selects all buttons, whether button elements directly or inputs of type button.
- `$(":checkbox")`: Selects all inputs of type checkbox.
- `$(":checked")`: Selects all checked inputs.
- `$(":contains('important')")`: Selects all elements containing the text `important`.
- `$(":disabled")`: Selects all inputs that are disabled.

- `$(":empty")`: Select all elements that have no children, not even a text node. This would include a node from `<img src='logo.png' height='120' width='100' alt='Widgets, Inc.' />" or "<p></p>", but not "<p>Hello.</p>" or the outer element of "<p><span></span></p>`.
- `$(":enabled")`: Selects all elements that are enabled.
- `$("p").eq(0)`: This gives two examples. First, all tags of a name may be found by calling their tag name; hence `$("li")` returns all `li` elements (regardless of the case in the source HTML). The `.eq()` function performs zero-based array indexing on the set. It is conceptually like how a JavaScript `$("p")[0]`. `$("p").eq(0)` returns a wrapped set containing one item: the first `p` element.
- `$(":even")`: Selects all even-numbered elements, but zero-based. Counter-intuitively, `$("table#directory tr:even")`, which says "Take the table with ID `directory` and return all even-numbered `tr` elements from it," will return the first, third, fifth, and so on, `tr` elements from the table if the table contains at least five table rows.
- `$(":file")`: Selects all inputs of type file.
- `$(":first-child")`: Selects all elements that are the first child of their parent. For example:

```
<html>
    <head>
        <link rel="stylesheet"
              type="text/css" href="/style.css" />
        <title>Welcome to our community!</title>
        <meta http-equiv="refresh" content="900" />
    </head>
    <body>
        <p>Welcome to our community! We offer:</p>
        <ul>
           <li>Experienced leadership.</li>
           <li>Well-furnished facilities.</li>
           <li>Good neighbors.</li>
        </ul>
    </body>
</html>
```

- `$(":first-child")` will return the head element, the link element, the `p` element, and the first `li` element. Note that this does not noisily and legalistically include every DOM text element. The text node containing "Good neighbors." is technically the first child of its `li` parent, but `$(":first-child")` returns a more useful, and conceptually cleaner to use, version of the first child.

- `.(":gt(2)")` will return all elements with (zero-based) index greater than two in the wrapped set. `$("p").(":gt(2)")` will return the fourth and subsequent `p` elements, or an empty wrapped set if there are not at least four `p` elements.
- `$(":has(a)")`: Returns all elements containing an anchor. `$("p:has(a)")` returns all `p` elements containing an `a` element.
- `$(":header")`: Returns all h1, h2, h3, and so on, elements.
- `$(":hidden")`: Returns all elements that are hidden.
- `$(":image")`: Returns all images.
- `$(":input")`: Returns all buttons, inputs, selects, and text areas.
- `$(":last-child")`: Like `$("first-child")`, but selects the last instead of first element.
- `.(":last")`: Returns the last element in a wrapped set if that set is non-empty; the Python equivalent to `$("p").(":last")` would be `paragraphs[-1]`.
- `.(":lt")`: Like `.(":gt")`, but selects elements less than the (zero-based) index. `$("p").(":lt(2)")` would return the first two out of any `p` elements.
- `.(":not(p a)")`: `$("p:not(p a)")` would return a matched set of all paragraphs that do not contain an `a` element.
- `$(":nth-child(3n)")`: Would return every element that is the *one-based* third child of its parent. As well as `$(":nth-child(3n)")`, `$(":nth-child(4n)")`, and so on, being allowed, there is a more intuitive, one-based `$(":nth-child(even)")` and `$(":nth-child(odd)")`, which more predictably have odd nth children starting with the first child and even with the second.

  The reason for this inconsistency is historical: other elements are zero-based in following JavaScript's and other languages' wide precedent, while this option is one-based in strictly following the CSS specification.
- `$(":odd")`: A selector that is counter-intuitively zero-based. `$("p:odd")` returns the second, fourth, sixth, and so on, paragraph elements if available.
- `$(":only-child")`: Selects all elements that are the only child of their parent.
- `$(":parent")`: Selects all nodes that are parents of other nodes, including text nodes. (Would return the `p` element for `<p>Hello, world!</p>`.)
- `$(":password")`: Selects all inputs of type password.
- `$(":radio")`: Selects all inputs of type radio.

- `$(":reset")`: Selects all inputs of type reset.
- `$(":selected")`: Returns all elements that are selected.
- `$(":submit")`: Returns all inputs of type submit.
- `$(":text")`: Returns all inputs of type text.
- `$(":visible")`: Returns all elements that are visible.

There are several remarks to be made here.

The first is that the selectors in these examples are (usually) given alone, but this is like words being listed alone in an old-fashioned paper dictionary. In English, there are a few things you can say with a single word, like "Stop!" but usually you say things by combining them, as we have done for a few examples. These selectors are powerful by themselves, but they are more powerful when viewed as words that make up sentences like `$("div.product ul:nth-child(odd)")`, which returns the first, third, fifth, and so on, `li` elements in each unordered list contained in a `div` of class `product`, such as one would use for tiger-striping unordered lists. Many of these selectors are useful by themselves, but they are intended, like pipable Unix command-line tools, to work well together and to be assembled like Lego bricks.

One of the first remarks one might make to someone learning Perl is, "You are not really thinking Perl until you are thinking dictionaries/hashes/associative arrays." If you are solving problems like they do in a C class, from the basic data types such as int, long[], char**, and void*, then you are missing one of the most important workhorses Perl has to offer (and Python, for that matter). And in similar fashion, *you are not really thinking jQuery until you are thinking wrapped sets as created by selectors like the examples above.* It's a foundational part of idiomatic use of the "virtual higher-level language" jQuery offers.

In some other libraries, and in unadorned JavaScript, you operate on the DOM one element at a time. If you want to operate on several elements, you still operate on them one at a time, but you do this several times in sequence. However, the basic unit of work in jQuery is the wrapped set; it may be a wrapped set of one, such as one may create by calling `$("#main:first-child")` or `$("div").eq(0)`, or one may end up getting it by calling `$("h2")` when the DOM contains exactly one `h2` header. However, even then it is missing something about jQuery to think of the set as simply a wrapper for an isolated, unitary element.

The list of the selectors above is something like a paper dictionary of nouns. We haven't yet discussed the verbs, or how to put them together in speech. Let's explore one example of including jQuery in simple Django Ajax. This example does not demonstrate jQuery's power and elegance yet. That is part of the goal in subsequent chapters, as they show Django Ajax put together using jQuery.

A closure-based example to measure clock skew

Let's put some things together in making a simple Ajax example. We will load a web page that will measure how long it takes to make a request from the server, and given a request that gives the time on the server, estimate clock skew between the server and the client. The server will give its answer in JSON, and we will do some DOM manipulations: we will use jQuery rather than an inline onclick-style attribute to register with the button's click event, and we will update the DOM without ever using innerHTML. And, for good measure, we will use a closure to make one incursion into the global namespace, so that if our code or some extension of it is reused, it will not overwrite other global variables.

First, let us make the template, for the sake of discussion starting from `base.html` as defined at the end of the last chapter. We will include it in the same directory as `base.html`, saved as `clockskew.html`:

```
{% extends "base.html" %}
{% block title %}Measure Clock Skew{% endblock title %}
{% block body_header_title %}Measure Clock Skew{% endblock body_
header_title %}
{% block head_css_page %}<style type="text/css">
<!--
.error
    {
    color: red;
    }
.success
    {
    font-weight: bold;
    }
// -->
</style>
{% endblock head_css_page %}
{% block footer_javascript_page %}<script language="JavaScript"
type="text/javascript" src="/static/js/json2.js"></script>
<script language="JavaScript" type="text/javascript"
        src="/static/js/clock_skew.js"></script>
{% endblock footer_javascript_page %}
{% block body_content %}<ul id="results"></ul>
<button id="button">Measure Clock Skew</button>
{% endblock body_content %}
```

Now this page could stand to be refactored on a couple of grounds. Firstly, it includes as page-specific `/static/js/json2.js`, which as a library for safer parsing of JavaScript presented as JSON should probably be included sitewide, or to go one step further, it should be concatenated with `/static/js/jquery.js` at least for sitewide deployment, and any other sitewide includes, so that only one JavaScript HTTP request slows things down. (As far as HTTP requests go, a 2 KB download plus another 2 KB download, especially if they are 2 KB JavaScript downloads, add up to more slowness in page rendering than one 4 KB download.) Secondly, one of the major principles of Django is DRY, and the fact that this page repeats "Measure Clock Skew" three times is an invitation for refactoring. This example and what follows deliberately have room left for refinements. (*Can you spot any further improvements to make?*)

We will also define a couple of models, one to serve up this template and one to serve JSON, adapt and extend the `urls.py` file, and write the Ajax to add behavior to the page. Here is the JavaScript `clock_skew.js` file:

```
var MeasureClockSkew = function()
    {
    var that = this;
    var lastButtonPress = new Date().getTime();
    var registerError = function(XMLHttpRequest, textStatus,
      errorThrown)
        {
        $("#results").append("<li class='error'>Error: " +
          textStatus + "</li>");
        $("#button").removeAttr("disabled");
        };
    var registerSuccess = function(data, textStatus, XMLHttpRequest)
        {
        try
            {
            var remote = JSON.parse(data).time;
            var halfway = (lastButtonPress +
              new Date().getTime()) / 2;
            var skew = (remote - halfway) / 1000;
            $("#results").append("<li class='success'>Estimated clock
              skew: <span class='measurement'>" + skew + "</span>
              seconds.</li>");
            }
        catch(error)
            {
            $("#results").append("<li class='error'>Error parsing
              JSON.</li>");
            }
        $("#button").removeAttr("disabled");
        };
    var buttonPress = function()
        {
        lastButtonPress = new Date().getTime();
        $("#button").attr("disabled", "disabled");
        $.ajax({data: "", dataType: "text", error: registerError,
          success: registerSuccess, type: "POST",
          url: "/time/json"});
        };
    return {
        buttonPress: buttonPress
        }
    } ( );
$("#button").click(MeasureClockSkew.buttonPress);
```

Before going further, we would like to make a few comments about double submission and race conditions. If our test script is deployed live on a faraway web server, with netlag we could fairly easily click the button twice before the response came back, and then it would calculate invalid data. The object, as a means of accounting for netlag, estimates that the server gave its timestamp halfway between when we submitted the click and when we received it, and if we clicked a button twice before the response came back, in addition to any inaccuracies in this estimation, it would combine the time the second click was made and when the first click's response came back, making the calculation corrupt. There are other ways this could have been dealt with; we could make a closure within a closure that would create a separate object for each click and allow overlapping trials with each end time matched to the corresponding start time. But here we have followed a much more generally applicable pattern from the e-commerce world, which is to disable the submit button so the user should not be able to generate overlapping clicks. This is another area where doing things right, even if it means doing double work, means that on the client side you prevent a second submission when that would not be in your visitor's best interests (you don't want your customers charged twice because they got impatient and pressed the button again), and on the server side you also take actions to prevent undesirable forms of double submission (remember, not all visitors have JavaScript enabled).

Furthermore, there is one other best practice worth mentioning: `this` is available only when an object is being constructed; but we can save a reference as `that`.

A Django model is a class that corresponds to a table in a database in Django's object-relational mapping. A Django model instance corresponds to a table row. The division of labor, or separation of concerns, is not exactly MVC, or model-view-controller, but MTV, model-template-view, where the model is a class of object that corresponds to a table in the database, a template is a designer-editable component that gets most HTML out of Python code, and a view is what renders a template or otherwise generates a loaded page. We will create two Django view methods to serve things up on the server side. One is like what we have seen before, and another is new but in principle self-explanatory. A view that serves up JSON is as follows:

```
#!/usr/bin/python/

import json
import time
from django.core import serializers
from django.http import HttpResponse
from django.shortcuts import render_to_response

def home(request):
    return render_to_response(u'clock_skew.html')

def timestamp(request):
    return HttpResponse(json.dumps({u'time': 1000 * time.time()}),
      mimetype=u'application/json')
```

We save both views in `clock_skew.py`, and then `edit urls.py`, so that after

```
    (r'^$', 'sample.views.home'),
```

we also have:

```
    (r'^time$', 'sample.clock_skew.home'),
    (r'^time/json$', 'sample.clock_skew.timestamp'),
```

This is a brief nutshell example of many, but not all, of the kinds of features we will use in our more in-depth case study. The astute reader may have noticed that this brief microcosm does not have the server storing information and saving the state for the user later. However, this does provide Ajax, jQuery, JSON, and some of the most foundational features that Django offers. As we move on we will take these features, incorporate Django models and database usage, and move into a more complex and more sophisticated usage of the features that are presented in this brief example.

Case study: A more in-depth application

In the following chapters, we will build a Django Ajax web application and use jQuery. The web application will be meant to work as a company's intranet employee photo directory, and we hope to put you in a position both to use our model application and customize it to your company's specific needs. In this test application, we will demonstrate both basic features and best practices in putting together a web application using our core technologies. The chapters in our case study will include the following sections.

Chapter 3: Validating Form Input on the Server Side

In this chapter, we will send an Ajax request to the server via jQuery, and validate it on the server side based on the principle that all input is guilty until proven innocent of being malicious, malformed, incomplete, or otherwise invalid. We will look at standard server validation approaches in light of usability practices and look for improvement.

Chapter 4: Server-side Database Search with Ajax

We will compare the merits of server-side and client-side searching, and look at the limitations of JavaScript, both in terms of language limitations, and in terms of client-side performance issues associated with doing too much in the client. We will look both at the merits of handling searching and other backend functions with the full power of a backend environment, and explore why, on the client side, we should work hard to be as lazy as possible in doing network-related work.

We might clarify that "lazy" here does not *specifically* refer to the programmer's virtue of a proactive laziness that tries to solve a problem once, correctly, rather than dash off a bad solution and then spend a lot of time cleaning up after a suboptimal solution. That is worth encouraging, but it is not what we are talking about here. "Lazy" refers, for instance, to Python's `xrange()`, a generator which yields integers one at a time as they are requested, rather than Python's `range()`, that builds a complete array immediately and takes significantly more memory for large ranges.

In our case, "lazy" refers in particular to not having the client try to anticipate user needs by fetching what might or might not be needed beforehand, but requesting the minimum necessary to meet user requests, only when requested. We will be exploring several approaches; "lazy" is one approach that we should definitely know about and be able to use.

Chapter 5: Signing Up and Logging into a Website Using Ajax

This chapter will introduce Django authentication facilities and account management, and explore how we can attractively handle the client side of authentication through Ajax client-side communication with the server and corresponding Ajax client-side updates.

Chapter 6: jQuery In-place Editing Using Ajax

In this chapter, we will show a way to use jQuery to make an in-place replacement of a table that allows in-place editing, which communicates with the server in the background, adding persistence to changes.

Chapter 7: Using jQuery UI Autocomplete in Django Templates

We will discuss jQuery's basic intention as having a useful core that's designed to invite plugins to the point of it not being uncommon for programmers to write plugins their first day doing jQuery. We will use autocomplete from the jQuery UI. We will then integrate this into Django templates and our project's user interface.

Chapter 8: Django ModelForm: a CSS Makeover

Django comes with a straightforward way to easily build forms from Django models. We will explore this feature and how to use it.

Chapter 9: Database and Search Handling

In this chapter, we will be showing "lazy" best practices in developing our application.

Chapter 10: Tinkering Around: Bugfixes, Friendlier Password Input, and a Directory That Tells Local Time

If you are interested in having an employee photo directory for your intranet, we not only want to provide an application but also help you have a starting point to create a customized application around your company's needs.

Chapter 11: Usability for Hackers

If you are reading this book, you may have some surprising strengths for usability. This chapter explores them. With this chapter we take a step back from our application and take a look at usability and the bedrock competencies hackers can leverage to do usability.

Appendix: Debugging Hard JavaScript Bugs

In this appendix, we take a look at the state of mind that is needed to debug difficult bugs.

Summary

The goal of these first two chapters has been to provide both a big picture and a sense for how things fit together. Ajax is a bit interdisciplinary; it involves server-side technologies (Django for us), client-side scripting and libraries (including jQuery for us), CSS, HTML, and the DOM, and the goal is very human in character. Ajax is interesting because it opens doors in user interface, usability, and user experience, and in that regard doing well with Ajax isn't just technical; it's also a bit like the arts and humanities.

We have covered the technical side of Django Ajax with jQuery in broad strokes, including:

- A tour of "Hello, world!" in Ajax, and how jQuery makes this an easier operation than library-free, "bare metal" JavaScript
- The basic facilities jQuery offers for Ajax
- A tour of jQuery selectors as an illustration of how jQuery offers a more Pythonic "virtual higher-level language"
- A minimal Django Ajax application with jQuery, with a discussion of what is going on
- An overview of the in-depth application to be covered in the upcoming chapters

These upcoming chapters will move from broad strokes to more in-depth competencies with specific technologies integrated into a more in-depth application.

Let's begin!