1
jQuery and Ajax Integration in Django

We will be working with the leading Python web framework, Django, on the server side, and jQuery-powered Ajax on the client side. During the course of this book, we will cover the basic technologies and then see them come together in an employee intranet photo directory that shares some Web 2.0 strengths.

There is more than one good JavaScript library; we will be working with jQuery, which has reached acceptance as a standard lightweight JavaScript library. It might be suggested that Pythonistas may find much to like in jQuery: jQuery, like Python, was carefully designed to enable the developer to get powerful results easily.

In this chapter, we will:

- Discuss Ajax as not a single technology but a technique which is overlaid on other technologies
- Cover the basic technologies used in Ajax JavaScript
- Cover "Hello, world!" in a Django kickstart
- Introduce the Django templating engine
- Cover how to serve up static content in Django

Overall, what we will be doing is laying a solid foundation and introducing the working pieces of Django Ajax to be explored in this book.

Ajax and the XMLHttpRequest object

Ajax is not a technology like JavaScript or CSS, but is more like an overlaid function. So, what exactly is that?

Human speech: An overlaid function

Human speech is an **overlaid function**. What is meant by this is reflected in the answer to a question: "What part of the human body has the basic job of speech?" The tongue, for one answer, is used in speech, but it also tastes food and helps us swallow. The lungs and diaphragm, for another answer, perform the essential task of breathing. The brain cannot be overlooked, but it also does a great many other jobs. All of these parts of the body do something more essential than speech and, for that matter, all of these can be found among animals that cannot talk. Speech is something that is *overlaid* over organs that are there in the first place because of something other than speech.

Something similar to this is true for **Ajax**, which is not a technology in itself, but something **overlaid** on top of other technologies. Ajax, some people say, stands for **Asynchronous JavaScript and XML**, but that was a retroactive expansion. JavaScript was introduced almost a decade before people began seriously talking about Ajax. Not only is it technically possible to use Ajax without JavaScript (one can substitute VBScript at the expense of browser compatibility), but there are quite a few substantial reasons to use **JavaScript Object Notation** (**JSON**) in lieu of heavy-on-the-wire **eXtensible Markup Language** (**XML**). Performing the *overlaid function* of Ajax with JSON replacing XML is just as eligible to be considered full-fledged Ajax as a solution incorporating XML.

Ajax: Another overlaid function

What exactly is this overlaid function?

Ajax is a way of using client-side technologies to talk with a server and perform partial page updates. Updates may be to all or part of the page, or simply to data handled behind the scenes. It is an alternative to the older paradigm of having a whole page replaced by a new page loaded when someone clicks on a link or submits a form. Partial page updates, in Ajax, are associated with **Web 2.0**, while whole page updates are associated with **Web 1.0**; it is important to note that "Web 2.0" and "Ajax" are not interchangeable. Web 2.0 includes more decentralized control and contributions besides Ajax, and for some objectives it may make perfect sense to develop an e-commerce site that uses Ajax but does not open the door to the same kind of community contributions as Web 2.0.

Some of the key features common in Web 2.0 include:

- Partial page updates with JavaScript communicating with a server and rendering to a page
- An emphasis on user-centered design
- Enabling community participation to update the website
- Enabling information sharing as core to what this communication allows

The concept of "partial page updates" may not sound very big, but part of its significance may be seen in an unintended effect. The original expectation of partial page updates was that it would enable web applications that were more responsive. The expectation was that if submitting a form would only change a small area of a page, using Ajax to just load the change would be faster than reloading the entire page for every minor change. That much was true, but once programmers began exploring, what they used Ajax for was not simply minor page updates, but making client-side applications that took on challenges more like those one would expect a desktop program to do, and the more interesting Ajax applications usually became slower. Again, this was not because you could not fetch part of the page and update it faster, but because programmers were trying to do things on the client side that simply were not possible under the older way of doing things, and were pushing the envelope on the concept of a web application and what web applications can do.

The technologies Ajax is overlaid on

Now let us look at some of the technologies where Ajax may be said to be *overlaid*.

JavaScript

JavaScript deserves pride of place, and while it is possible to use VBScript for Internet Explorer as much more than a proof of concept, for now if you are doing Ajax, it will almost certainly be Ajax running JavaScript as its engine. Your application will have JavaScript working with `XMLHttpRequest`, JavaScript working with HTML, XHTML, or HTML5; JavaScript working with the DOM, JavaScript working with CSS, JavaScript working with XML or JSON, and perhaps JavaScript working with other things.

While addressing a group of Django developers or Pythonistas, it would seem appropriate to open with, "I share your enthusiasm." On the other hand, while addressing a group of JavaScript programmers, in a few ways it is more appropriate to say, "I feel your pain." JavaScript is a language that has been discovered as a gem, but its warts were enough for it to be largely unappreciated for a long time. "Ajax is the gateway drug to JavaScript," as it has been said—however, JavaScript needs a gateway drug before people get hooked on it. JavaScript is an excellent language and a terrible language rolled into one.

Before discussing some of the strengths of JavaScript—and the language does have some truly deep strengths—I would like to say "I feel your pain" and discuss two quite distinct types of pain in the JavaScript language.

The first source of pain is some of the language decisions in JavaScript:

- The Wikipedia article says it was designed to resemble Java but be easier for non-programmers, a decision reminiscent of SQL and COBOL.
- The Java programmer who finds the C-family idiom of `for(i = 0; i < 100; ++i)` available will be astonished to find that the functions are clobbering each other's assignments to `i` until they are explicitly declared local to the function by declaring the variables with `var`. There is more pain where that came from.

The following two functions will not perform the naively expected mathematical calculation correctly; the assignments to `i` and the result will clobber each other:

```
function outer()
    {
    result = 0;
    for(i = 0; i < 100; ++i)
        {
        result += inner(i);
        }
    return result
    }

function inner(limit)
    {
    result = 0;
    for(i = 0; i < limit; ++i)
        {
        result += i;
        }
    return result;
    }
```

The second source of pain is quite different. It is a pain of inconsistent implementation: the pain of, "Write once, debug everywhere." Strictly speaking, this is not JavaScript's fault; browsers are inconsistent. And it need not be a pain in the server-side use of JavaScript or other non-browser uses. However, it comes along for the ride for people who wish to use JavaScript to do Ajax. Cross-browser testing is a foundational practice in web development of any stripe; a good web page with semantic markup and good CSS styling that is developed on Firefox will usually look sane on Internet Explorer (or vice versa), even if not quite pixel-perfect. But program directly for the JavaScript implementation on one version of a browser, and you stand rather sharp odds of your application not working at all on another browser. The most important object by far for Ajax is the `XMLHttpRequest` and not only is it not the case that you may have to do different things to get an XMLHttpRequest in different browsers or sometimes different (common) versions of the same browser, and, even when you have code that will get an `XMLHttpRequest` object, the objects you have can be incompatible so that code that works on one will show strange bugs for another. Just because you have done the work of getting an `XMLHttpRequest` object in all of the major browsers, it doesn't mean you're home free.

Before discussing some of the strengths of the JavaScript language itself, it would be worth pointing out that a good library significantly reduces the second source of pain. Almost any sane library will provide a single, consistent way to get `XMLHttpRequest` functionality, and consistent behavior for the access it provides. In other words, one of the services provided by a good JavaScript library is a much more uniform behavior, so that you are programming for only one model, or as close as it can manage, and not, for instance, pasting conditional boilerplate code to do simple things that are handled differently by different browser versions, often rendering surprisingly different interpretations of JavaScript. We will be using the jQuery library in this book as a standard, well-designed, lightweight library. Many of the things we will see done well as we explore jQuery are also done well in other libraries.

We previously said that JavaScript is an excellent language and a terrible language rolled into one; what is to be said in favor of JavaScript? The list of faults is hardly all that is wrong with JavaScript, and saying that libraries can dull the pain is not itself a great compliment. But in fact, something much stronger can be said for JavaScript: *If you can figure out why Python is a good language, you can figure out why JavaScript is a good language.*

I remember, when I was chasing pointer errors in what became 60,000 lines of C, teasing a fellow student for using Perl instead of a real language. It was clear in my mind that there were interpreted scripting languages, such as the bash scripting that I used for minor convenience scripts, and then there were *real* languages, which were compiled to machine code. I was sure that a real language was identified with being compiled, among other things, and that power in a language was the sort of thing C traded in. (I wonder why he didn't ask me if he wasn't a real programmer because he didn't spend half his time chasing pointer errors.) Within the past year or so I've been asked if "Python is a real programming language or is just used for scripting," and something similar to the attitude shift I needed to appreciate Perl and Python is needed to properly appreciate JavaScript.

The name "JavaScript" is unfortunate; like calling Python "Assembler Kit", it's a way to ask people not to see its real strengths. (Someone looking for tools for working on an assembler would be rather disgusted to buy an "Assembler Kit" and find Python inside. People looking for Java's strengths in JavaScript will almost certainly be disappointed.)

JavaScript code may look like Java in an editor, but the resemblance is a façade; besides Mocha, which had been renamed LiveScript, being renamed to JavaScript just when Netscape was announcing Java support in web browsers, it is has been described as being descended from NewtonScript, Self, Smalltalk, and Lisp, as well as being influenced by Scheme, Perl, Python, C, and Java. What's under the Java façade is pretty interesting. And, in the sense of the simplifying "façade" design pattern, JavaScript was marketed in a way almost guaranteed not to communicate its strengths to programmers. It was marketed as something that nontechnical people could add snippets of, in order to achieve minor, and usually annoying, effects on their web pages. It may not have been a toy language, but it sure was dressed up like one.

Python may not have functions clobbering each other's variables (at least not unless they are explicitly declared global), but Python and JavaScript are both multiparadigm languages that support object-oriented programming, and their versions of "object-oriented" have a lot in common, particularly as compared to (for instance) Java. In Java, an object's class defines its methods and the type of its fields, and this much is set in stone. In Python, an object's class defines what an object starts off as, but methods and fields can be attached and detached at will. In JavaScript, classes as such do not exist (unless simulated by a library such as Prototype), but an object can inherit from another object, making a prototype and by implication a prototype chain, and like Python it is dynamic in that fields can be attached and detached at will. In Java, the `instanceof` keyword is important, as are class casts, associated with strong, static typing; Python doesn't have casts, and its `isinstance()` function is seen by some as a mistake, hence the blog posting "`isinstance()` considered harmful" at `http://www.canonical.org/~kragen/isinstance/`.

The concern is that Python, like JavaScript, is a duck-typing language: *If it looks like a duck, and it quacks like a duck, it's a duck!* In a duck-typing language, if you write a program that polls weather data, and there's a `ForecastFromScreenscraper` object that is several years old and screenscrapes an HTML page, you should be able to write a `ForecastFromRSS` object that gets the same information much more cleanly from an RSS feed. You should be able to use it as a drop-in replacement as long as you have the interface right. That is different from Java; at least if it were a `ForecastFromScreenscraper` object, code would break immediately if you handed it a `ForecastFromRSS` object. Now, in fairness to Java, the "best practices" Java way to do it would probably separate out an `IForecast` interface, which would be implemented by both `ForecastFromScreenscraper` and later `ForecastFromRSS`, and Java has ways of allowing drop-in replacements *if* they have been explicitly foreseen and planned for. However, in duck-typed languages, the reality goes beyond the fact that if the people in charge designed things carefully and used an interface for a particular role played by an object, you can make a drop-in replacement. In a duck-typed language, you can make a drop-in replacement for things that the original developers never imagined you would want to replace.

JavaScript's reputation is changing. More and more people are recognizing that there's more to the language than design flaws. More and more people are looking past the fact that JavaScript is packaged like Java, like packaging a hammer to give the impression that it is basically like a wrench. More and more people are looking past the silly "toy language" Halloween costume that JavaScript was stuffed into as a kid.

One of the ways good programmers grow is by learning new languages, and JavaScript is not just the gateway to mainstream Ajax; it is an interesting language in itself. With that much stated, we will be making a carefully chosen, selective use of JavaScript, and not make a language lover's exploration of the JavaScript language, overall. Much of our work will be with the jQuery library; if you have just programmed a little "bare JavaScript", discovering jQuery is a bit like discovering Python, in terms of a tool that cuts like a hot knife through butter. It takes learning, but it yields power and interesting results soon as well as having some room to grow.

XMLHttpRequest

The `XMLHttpRequest` object is the reason why the kind of games that can be implemented with Ajax technologies do not stop at clones of Tetris and other games that do not know or care if they are attached to a network. They include massive multiplayer online role-playing games where the network is the computer. Without having something like `XMLHttpRequest`, "Ajax chess" would probably mean a game of chess against a chess engine running in your browser's JavaScript engine; with `XMLHttpRequest`, "Ajax chess" is more likely man-to-man chess against another human player connected via the network. The `XMLHttpRequest` object is the object that lets Gmail, Google Maps, Bing Maps, Facebook, and many less famous Ajax applications deliver on Sun's promise: the network *is* the computer.

There are differences and some incompatibilities between different versions of `XMLHttpRequest`, and efforts are underway to advance "level-2-compliant" `XMLHttpRequest` implementations, featuring everything that is expected of an `XMLHttpRequest` object today and providing further functionality in addition, somewhat in the spirit of level 2 or level 3 CSS compliance. We will not be looking at level 2 efforts, but we will look at the baseline of what is expected as standard in most `XMLHttpRequest` objects.

The basic way that an `XMLHttpRequest` object is used is that the object is created or reused (the preferred practice usually being to reuse rather than create and discard a large number), a callback event handler is specified, the connection is opened, the data is sent, and then when the network operation completes, the callback handler retrieves the response from `XMLHttpRequest` and takes an appropriate action.

A bare-bones `XMLHttpRequest` object can be expected to have the following methods and properties.

Methods

A bare-bones `XMLHttpRequest` object can be expected to have the following methods:

1. `XMLHttpRequest.abort()`

   This cancels any active request.

2. `XMLHttpRequest.getAllResponseHeaders()`

   This returns all HTTP response headers sent with the response.

3. `XMLHttpRequest.getResponseHeader(headerName)`

   This returns the requested header if available, or a browser-dependent false value if the header is not defined.

4. `XMLHttpRequest.open(method, URL)`,
   `XMLHttpRequest.open(method, URL, asynchronous)`,
   `XMLHttpRequest.open(method, URL, asynchronous, username)`,
   `XMLHttpRequest.open(method, URL, asynchronous, username, password)`

   The `method` is `GET`, `POST`, `HEAD`, or one of the other less frequently used methods defined for HTTP.

   The `URL` is the relative or absolute URL to fetch. As a security measure for JavaScript running in browsers on trusted internal networks, a **same origin policy** is in effect, prohibiting direct access to servers other than one the web page came from. Note that this is less restrictive than it sounds, as it is entirely permissible for the server to act as a proxy for any server it has access to: for developers willing to undertake the necessary chores, other sites on the public internet are "virtually accessible".

   The `asynchronous` variable defaults to `true`, meaning that the method call should return quickly in most cases, instead of waiting for the network operation to complete. Normally this default value should be preserved. Among other problems, setting it to `false` can lock up the visitor's browser.

   The last two arguments are the username and password as optionally specified in HTTP. If they are not specified, they default to any username and password defined for the web page.

5. `XMLHttpRequest.send(content)`

   Content can be a string or a reference to a document.

Properties

A bare-bones `XMLHttpRequest` object can be expected to have the following properties:

1. `XMLHttpRequest.onreadystatechange`,
   `XMLHttpRequest.readyState`

   In addition to the provided methods, the reference to one other method is supplied by the developer as a property, `XMLHttpRequest.onreadystate-change`, which is called without argument each time the ready state of `XMLHttpRequest` changes. An `XMLHttpRequest` object can have five ready states:

   - Uninitialized, meaning that `open()` has not been called.
   - Open, meaning that `open()` has been called but `send()` has not.
   - Sent, meaning that `send()` has been called, and headers and status are available, but the response is not yet available.

- Receiving, meaning that the response is being downloaded and `responseText` has the portion that is presently available.
- Loaded, meaning that the network operation has completed. If it has completed successfully (that is, the HTTP status stored in `XMLHttpRequest.status` is `200`), this is when the web page would be updated based on the response.

2. `XMLHttpRequest.responseText`, `XMLHttpRequest.responseXML`

   The text of the response. It is important to note that while the name "*XML*HttpRequest" is now very well established, and it was originally envisioned as a tool to get *XML*, the job done today is quite often to get *text* that may or may not happen to be XML. While there have been problems encountered with using `XMLHttpRequest` to fetch raw binary data, the `XMLHttpRequest` object is commonly used to fetch not only XML but XHTML, HTML, plain text, and JSON, among others. If it were being named today, it would make excellent sense to name it "*Text*HttpRequest." Once the request reaches a ready state of 4 ("loaded"), the `responseText` field will contain the text that was served up, whether the specific text format is XML or anything else. In addition, if the format does turn out to be XML, the `responseXML` field will hold a parsed XML document.

3. `XMLHttpRequest.status`, `XMLHttpRequest.statusText`

   The `status` field contains the HTTP code, such as 200 for OK; the `statusText` field has a short text description, like OK. The callback event handler should ordinarily check `XMLHttpRequest.readyState` and wait before acting on server-provided data until the `readyState` is 4. In addition, because there could be a server error or a network error, the callback will check whether the status is 200 or something else: a code like 4xx or 5xx in particular needs to be treated as an error. If the server-response has been transmitted successfully, the `readyState` will be 4 and the `status` will be 200.

This is the basic work that needs to be done for the `XMLHttpRequest` side of Ajax. Other frameworks may simplify this and do much of the cross-browser debugging work for you; we will see in the next chapter how jQuery simplifies this work. But this kind of task is something you will need to have done with any library, and it's worth knowing what's behind the simplified interfaces that jQuery and other libraries provide.

HTML/XHTML

HTML and XHTML make up the bedrock markup language for the web. JavaScript and CSS were introduced in relation to HTML; perhaps some people are now saying that JavaScript is a very interesting language independent of web browsers and using standalone interpreters such as SpiderMonkey and Rhino. However, HTML was on the scene first and other players on the web exist in relation to HTML's story. Even when re-implemented as XHTML, to do HTML's job while potentially making much more sense to parsers, a very early web page, the beginning of the source at `http://www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html`, is still quite intelligible:

```
<HEADER>
<TITLE>The World Wide Web project</TITLE>
<NEXTID N="55">
</HEADER>
<BODY>
<H1>World Wide Web</H1>The WorldWideWeb (W3)
is a wide-area
<A NAME=0 HREF="WhatIs.html">
hypermedia</A> information retrieval
initiative aiming to give universal
access to a large universe of documents.<P>
Everything there is online about
W3 is linked directly or indirectly
to this document, including an
<A NAME=24 HREF="Summary.html">executive
summary</A> of the project,
<A NAME=29 HREF="Administration/Mailing/Overview.html">
Mailing lists</A> ,
<A NAME=30 HREF="Policy.html">Policy</A> , November's
<A NAME=34 HREF="News/9211.html">W3  news</A> ,
<A NAME=41 HREF="FAQ/List.html">Frequently Asked Questions
</A> .
<DL>
…
```

At the time of this writing, HTML 5 is taking shape but is not "out in the wild", and so there are no reports of how the shoe feels after the public has worn it for a while. Code in this book, where possible, will be written in XHTML 1.0 Strict. Depending on your situation, this may or may not be the right decision for you; if you are working with an existing project, the right HTML/XHTML is often the one that maintains consistency within the project.

XML

eXtensible Markup Language (**XML**) is tied to an attempt to clean up early HTML. At least in earliest forms, HTML was a black sheep among specific markup languages derived from the generalized and quite heavy **Standard Generalized Markup Language** (**SGML**). Forgiving web browsers meant, in part, that early web hobbyists could write terrible markup and it would still display well in a browser. The amount of terrible markup on the web was not just an issue for purists; it meant that making a parser that could make sense of early "Wild West" web pages in general was a nearly impossible task. XML is vastly simplified from SGML, but it provides a generic space where an HTML variant, XHTML, could pick up the work done by HTML but not present parsers with unpredictable tag soup. XHTML could be described as HTML brought back into the fold, still good for doing web development, but without making machine interpretation such a hopeless cause. Where early HTML was developed with browsers that were meant to be forgiving, XML requested draconian error handling, and validated XML or XHTML documents are documents that can be parsed in a sensible way.

XML works for exchanging information, and it works where many of its predecessors had failed: it provides interoperability between different systems after a long history of failed attempts at automating B2B communication and failed attempts at automated conversion between text data formats. Notwithstanding this, it is a heavy and verbose solution, with a bureaucratic ambiance, compared in particular to a lean, mean JSON. XML-based approaches to data storage and communication are increasingly critiqued in discussions on the web. If you have a reasonable choice between XML and JSON, we suggest that you seriously consider JSON.

JSON

JavaScript Object Notation (**JSON)** is a brilliantly simple idea. While formats like XML, ReStructuredText, and so on share the assumption that "if you're going to parse this from your language, your language will need to have a parser added," JSON simply takes advantage of how an object would be specified in JavaScript, and clarifies a couple of minor points to make JSON conceptually simpler and cross-browser friendly. JSON is clear, simple, and concise enough that not only is it a format of choice for JavaScript, but it is gaining traction in other languages, and it is being used for communication between languages that need a (simple, added) parser to parse JSON. The other languages can't use `eval()` to simply run JSON, and in JavaScript you should have JSON checked to make sure it does not contain malicious JavaScript you should not `eval()`. However, JSON is turning out to have a much broader impact than the initial "in communicating with JavaScript, just give it code to declare the object being communicated that can simply be evaluated to construct the object."

CSS

Cascading Style Sheets (**CSS**) may have introduced some new possibilities for presentation, but quite a lot of presentation was already possible beforehand. CSS did not so much add styling capabilities, as it added good engineering *to* styling (good engineering is the essence of "separating presentation from content"), and make the combination of semantic markup *and* attractive appearance a far more attainable goal. It allows parlor tricks such as in-place rebranding of websites: making changes in images and changing one stylesheet is, at least in principle, enough to reskin an extensive website without touching a single character of its HTML/XHTML markup. In Ajax, as for the rest of the web, the preferred practice is to use semantic, structural markup, and then add styles in a stylesheet (not inline) so that a particular element, optionally belonging to the right class or given the right ID, will have the desired appearance. Tables are not deprecated but should be used for semantic presentation of tabular data where it makes sense to use not only a `td` but a `th` as well. What is discouraged is using the side effect that tables can position content that is not, semantically speaking, tabular data.

The DOM

As far as direct human browsing is concerned, HTML and associated technologies are vehicles to deliver a pickled **Document Object Model** (**DOM**), and nothing more. In this respect, HTML is a means to an end: the DOM is the "deserialized object," or better, the "live form" of what we really deliver to people. HTML may help provide a complete blueprint, and the "complete blueprint" is a means to the "fully realized building." This is why solving Ajax problems on the level of HTML text are like answering the wrong question, or at least solving a problem on the wrong level. It is like deciding that you want a painting hung on a wall of a building, and then going about getting it by adding the painting to the blueprint and asking construction personnel to implement the specified change. It may be better to hang the painting on the wall directly, as is done in Ajax DOM manipulations.

`document.write()` and `document.getElementById().innerHTML()` still have a place in web development. It is a sensible optimization to want a static, cacheable HTML/XHTML file include that will only be downloaded once in the usual multi-page visit. A JavaScript include with a series of `document.write()` may be the least Shanghaiing you can do to technologies and still achieve that goal. But this is *not* Ajax; it is barely JavaScript, and this is not where we should be getting our bearings. In Ajax, a serious alternative to this kind of solution for altering part of a web page is with the DOM.

As the book progresses, we will explore Ajax development that works with the DOM.

iframes and other Ajax variations

Ajax includes several variations; **Comet** for instance, is a variation on standard Ajax in which either an `XMLHttpRequest` object's connection to a server is kept open and streaming indefinitely, or a new connection is opened whenever an old one is closed, creating an Ajax environment in which the server as well as the client can push material. This is used, for instance, in some instant messaging implementations. One much more essential Ajax variation has to do with loading documents into seamlessly integrated iframes instead of making DOM manipulations to a single, frame-free web page.

If you click around on the page for a Gmail account, you will see partial page refreshes that look consistent with Ajax DOM manipulations: what happens when you click on **Compose Mail**, or a filter, or a message subject, looks very much like an Ajax update where the Gmail web application talks with the server if it needs to, and then updates the DOM in accordance with your clicks. However, there is one important difference between Gmail's behavior and a similar Ajax clone that updates the DOM for one frameless web page: what happens when you click the browser "Back" button. Normally, if you click on a link, you trigger an Ajax event but not a whole page refresh, and Ajax optionally communicates with a server and updates some part of the DOM. This does not register in the browser's history, and hitting the **Back** button would not simply reset the last Ajax partial page update. If you made an Ajax clone of Gmail that used DOM manipulations instead of seamlessly integrated iframes, there would be one important difference in using the clone: hitting **Back** would do far more than reverse the last DOM manipulation. *It would take you back to the login or load screen.* In Gmail, the browser's **Back** button works with surgical accuracy, and the reason it can do something much better than take you back to the login screen is that Gmail is carefully implemented with iframes, and every change that the **Back** button can undo is implemented by a fresh page load in one of the seamlessly integrated iframes. That creates browsing history.

For that matter, a proof of concept has been created for an Ajax application that does not use client-side scripting or programming, instead using, on the client side, a system of frames/iframes, targets, links, form submissions, and meta refresh tags in order to perform partial page updates. Whether this variant technique lends itself to creating graceful alternatives to standard Ajax implementations, or is only a curiosity merely lending itself to proofs of concept, it is in principle possible to make an Ajax application that loses nothing if a visitor's browser has turned off scripting completely.

Comet and iframes are two of many possible variations on the basic Ajax technique; what qualifies as Ajax is more a matter of Python- or JavaScript-style duck-typing than Java-style static typing. "Asynchronous JavaScript and XML" describes a reference example more than a strict definition, and it is not appropriate to say "if

you replace XML with JSON then, by definition, it isn't really Ajax." This is a case of, "the proof of the pudding is in the eating," not what technologies or even techniques are in the kitchen.

JavaScript/Ajax Libraries

This book advocates taking advantage of libraries, and as a limitation of scope focuses on jQuery. If you only learn one library, or if you are starting with just one library, jQuery is a good choice, and it is widely used. It is powerful, but it is also a much easier environment to get started in than some other libraries; in that way, it is somewhat like Python. However, it is best not to ask, "Which one library is best?" but "Which library or libraries are the right tools for this job?", and it is common real-world practice to use more than one library, possibly several.

JavaScript libraries offer several advantages. They can reduce chores and boilerplate code, significantly lessening the pain of JavaScript, and provide a more uniform interface. They can also provide (for instance) ready-made widgets; we will be working with a jQuery slider later on in this book. And on a broad scale, they can let the JavaScript you write be higher-level and a little more Pythonic.

Server-side technologies

Many of the "usual suspects" in client-side technologies have been mentioned. The list of client-side technologies is generally constrained by what is available in common web browsers; the list of available server-side technologies is only constrained by what will work on the server, and any general-purpose programming language *can* do the job. The question on the server is not "What is available?" but "Which option would you choose?" Python and Django make an excellent choice of server-side technology, and we will work with them in this book.

A look at Django

Django's developers call it "the web framework for perfectionists with deadlines," and it is one of the most popular Python web frameworks, perhaps the most popular. In contrast to the MVC pattern, which separates concerns into Model, View, and Controller, it could be described as an MTV pattern, which separates concerns into Model, Template, and View. The **Model** is a class that ties into an ORM where instances correspond to rows in the table but act and feel like Python objects. The **Template** is a system designed to be easy for non-Python developers (though easy for Pythonistas too), and limits the extent to which HTML needs to be sprinkled throughout the Python source. The **View** is a function that renders, in most cases, from a template. Let's look at a kickstart example of Django in action.

Django templating kickstart

Let us briefly go through how to install Django, create a sample project, and create and use a basic template that can serve as a basis for further tinkering.

Django installation instructions are at `http://docs.djangoproject.com/en/dev/intro/install/`; for Ubuntu, for instance, you will want to run `sudo apt-get install python-django`.

Once you have Django installed, create a project named `sample`:

```
django-admin.py startproject sample
```

Go into the `sample` directory, and create the directory `templates`. Enter the `templates` directory.

Create a template file named `index.html` containing the following template:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US"
lang="en-US">
  <head>
    <title>{% block title %}Hello, world!{% endblock title %}
    </title>
      <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8" />
  </head>
  <body>
    {% block body %}
      <h1>{% block heading %}Hello, world!
      {% endblock heading %}</h1>
      {% block content %}<p>Greetings from the Django
        templating engine!</p>{% endblock content %}
    {% endblock body %}
  </body>
</html>
```

Go up one level to the `sample` directory and edit the `urls.py` file so that the first line after `urlpatterns = patterns('',` is:

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

Then create the `views.py` file containing the following:

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

from django.shortcuts import render_to_response

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

Edit the `settings.py` file, and add:

```
os.path.join(os.path.dirname(__file__), "templates"),
```

right after:

```
TEMPLATE_DIRS = (
```

Then, from the command line, run:

```
python manage.py runserver
```

This makes the server accessible to your computer only by entering the URL `http://localhost:8080/` in your web browser.

If you are in a protected environment behind a firewall, appropriate NATting, or the like, you can make the development server available to the network by running:

```
python manage.py runserver 0.0.0.0:8080
```

There is one point of clarification we would like to make clear. Django is packaged with a minimal, single-threaded web server that is intended to be just enough to start exploring Django in a development environment. Django's creators are attempting to make a good, competitive web framework and not a good, competitive web server, and *the development server has never undergone a security audit.* The explicit advice from Django's creators is: when deploying, use a good, serious web server; they also provide instructions for doing this.

A more complete glimpse at Django templating

Before further exploring technical details, it would be worth taking a look at the opinions and philosophy behind the Django templating language, because an understandable approach of, "Oh, it's a general purpose programming language used for templating," is a recipe for needless frustration and pain. The Django developers themselves acknowledge that their opinions in the templating language are one just opinion in an area where different people have different opinions, and you are welcome to disagree with them if you want. If you don't like the templating system that Django comes with, Django is designed to let you use another. But it is worth understanding what exactly the philosophy is behind the templating language; even if this is not the only philosophy one could use, it is carefully thought out.

The Django templating language is intended to foster the separation of presentation and logic. In its design decisions, both large and small, Django's templating engine is optimized primarily for designers to use for designing, rather than programmers to use for programming, and its limitations are almost as carefully chosen as the features it provides. Unlike ASP, JSP, and PHP, it is not a programming language interspersed with HTML. It provides enough power for presentation, and is intended *not* to provide enough power to do serious programming work where it doesn't belong (in the Django opinion), and is simple enough that some non-programmers can pick it up in a day. For a programmer, the difficulty of learning the templating basics is comparable to the difficulty of simple HTML or SQL: it is simple, and a good bit easier to learn than wrapping your arms around a regular programming language. Some programmers like it immediately, but there are some who started by asking, "Why doesn't the templating language just let you mix Python and HTML?" and after playing with it, found themselves saying, "This isn't what I would have come up with myself, but I really, really like it."

Additional benefits include it being *fast* (most of the work is done by a single regular expression call, and the founders talk about disabling caching because it wasn't as fast as the template rendering), *secure* (it is designed so that it can be used by untrusted designers without allowing a malicious designer to execute arbitrary code), and *versatile* enough to generate whatever text format you want: plain text, HTML, XML, XHTML, JSON, JavaScript, CSV, ReStructuredText, and so on. We will be using it to generate web pages and JSON, but Django's templating language is a general-purpose text templating solution.

Following the Django site's lead, let us use a template intended as an example of how one might begin a base template for a site, then start to walk through its contents, and then look at some of how it could be used and parts overridden to create a specific document.

This renders as follows, if we strip out blank lines:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US"
lang="en-US">
  <head>
    <title></title>
    <link rel="icon" href="/static/favicon.ico"
          type="x-icon" />
    <link rel="shortcut icon" href="/static/favicon.ico"
          type="x-icon" />
    <meta http-equiv="Content-Type" content="text/html;
          charset=UTF-8" />
    <meta http-equiv="Content-Language" value="en-US" />
```

```
    <link rel="stylesheet" type="text/css"
          href="/static/css/style.css" />
  </head>
  <body>
    <div id="sidebar">
    </div>
    <div id="content">
      <div id="header">
        <h1></h1>
      </div>
    </div>
    <div id="footer">
    </div>
  </body>
    <script language="JavaScript" type="text/javascript"
            src="/static/js/jquery.js"></script>
</html>
```

Let us unwrap what is going on here; there is more to the template than how it renders to this page, but let us start with that much.

The `{% block dtd %}` style tags begin, or the case of `{% endblock dtd %}` end, a semantic block of text that can be left untouched or can be replaced. In the case of this one template, the effect is to strip them out like comments, but they will yield benefits later on, much like semantic HTML markup with CSS yields benefits later on.

Django templating reflects a choice to go with hooks rather than includes because hooks provide the more versatile solution. The "beginner's mistake" version of a header to include might be something like the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US"
lang="en-US">
  <head>
    <title>Welcome to my site!</title>
    <link rel="icon" href="/favicon.ico" type="x-icon" />
    <link rel="shortcut icon" href="/favicon.ico"
          type="x-icon" />
    <meta http-equiv="Content-Type" content="text/html;
          charset=UTF-8" />
    <meta http-equiv="Content-Language" value="en-US" />
    <link rel="stylesheet" type="text/css"
          href="/css/style.css" />
  </head>
  <body>
```

This solution could be continued by adding a sidebar, but there is a minor problem, or at least it seems minor at first: there are bits of this header that are not generic. For a serious site, having every page titled, "Welcome to my site!" would be an embarrassment. The language is declared to be "en-US", meaning U.S. English, which is wonderful if the entire site is in U.S. English, but if it expands to include more than U.S. English content, hardcoding "en-US" will be a problem. If the only concern is to accurately label British English, then the more expansive "en" could be substituted in, but hardcoding "en-US" and "en" are equally unhelpful if the site expands to feature a section in Russian. This header does not include other meta tags that might be desirable, such as "description", which is ideally written for a specific page and not done as site-wide boilerplate. Including the header verbatim solves a problem, but it doesn't provide a very flexible solution.

The previous example builds in hooks. It does specify:

```
{% block html_tag %}<html xmlns="http://www.w3.org/1999/xhtml" xml:
lang="en-US" lang="en-US">{% endblock html_tag %}
```

If not overridden, this will render as:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US"
lang="en-US">
```

However, in a template that extends this, by having `{% extends "base.html" %}` as its opening tag, if the base is loaded as `base.html`, then:

```
{% block html_tag %}<html xmlns="http://www.w3.org/1999/xhtml" xml:
lang="en-GB" lang="en-GB">{% endblock html_tag %}
```

will render:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-GB"
lang="en-GB">
```

And Russian may be declared in the same way:

```
{% block html_tag %}<html xmlns="http://www.w3.org/1999/xhtml" xml:
lang="ru-RU" lang="ru-RU">{% endblock html_tag %}
```

will render:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="ru-RU"
lang="ru-RU">
```

The template as given does specify "en-US" more than once, but each of these is inside a block that can be overridden to specify another language.

We define initial blocks. First, the DTD:

```
{% block dtd %}<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//
EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
{% endblock dtd %}
```

Then, the HTML tag:

```
{% block html_tag %}<html xmlns="http://www.w3.org/1999/xhtml" xml:
lang="en-US" lang="en-US">{% endblock html_tag %}
```

Then we define the head, with the title and favicon:

```
{% block head %}<head>
  <title>{% block title %}{{ page.title }}
    {% endblock title %}</title>
  {% block head_favicon %}<link rel="icon"
    href="/static/favicon.ico" type="x-icon" />
  <link rel="shortcut icon" href="/static/favicon.ico"
    type="x-icon" />{% endblock head_favicon %}
```

Then we define hooks for meta tags in the head. We define a Content-Type of UTF-8; this is a basic point so that non-ASCII content will display correctly:

```
  {% block head_meta %}
    {% block head_meta_author%}{% endblock head_meta_author %}
    {% block head_meta_charset %}
      <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8" />
    {% endblock head_meta_charset %}
    {% block head_meta_contentlanguage %}
      <meta http-equiv="Content-Language" value="en-US" />
    {% endblock head_meta_contentlanguage %}
    {% block head_meta_description %}
    {% endblock head_meta_description %}
    {% block head_meta_keywords %}
    {% endblock head_meta_keywords %}
    {% block head_meta_othertags %}
    {% endblock head_meta_othertags %}
    {% block head_meta_refresh %}
    {% endblock head_meta_refresh %}
    {% block head_meta_robots %}
    {% endblock head_meta_robots %}
  {% endblock head_meta %}
```

We declare a block to specify an RSS feed for the page:

```
  {% block head_rss %}{% endblock head_rss %}
```

We add hooks for CSS, both at the site level, and section, and page. This allows a fairly fine granularity of control:

```
{% block head_css %}
  {% block head_css_site %}
    <link rel="stylesheet" type="text/css"
          href="/static/css/style.css" />
  {% endblock head_css_site %}
  {% block head_css_section %}
  {% endblock head_css_section %}
  {% block head_css_page %}{% endblock head_css_page %}
{% endblock head_css %}
```

We add section- and page-specific header information:

```
{% block head_section %}{% endblock head_section %}
{% block head_page %}{% endblock head_page %}
```

Then we close the head and open the body:

```
</head>{% endblock head %}
  {% block body %}
  <body>
```

We define a sidebar block, with a hook to populate it:

```
<div id="sidebar">
  {% block body_sidebar %}{% endblock body_sidebar %}
</div>
```

We define a block for the main content area:

```
<div id="content">
{% block body_content %}
```

For the header of the main content area, we define a banner hook, and a header for the page's title, should such be provided. (If none is provided, there is no crash or error; the empty string is displayed for `{{ page.title }}`.)

```
  <div id="header">
  {% block body_header %}
    {% block body_header_banner %}
    {% endblock body_header_banner %}
    {% block body_header_title %}<h1>
      {{ page.title }}</h1>
    {% endblock body_header_title %}
```

We define a breadcrumb, which is one of many small usability touches that can be desirable:

```
      {% block body_header_breadcrumb %}
        {{ page.breadcrumb }}
      {% endblock body_header_breadcrumb %}
    {% endblock body_header %}
    </div>
```

We add a slot for announcements, then the body's main area, and then close the block and `div`:

```
      {% block body_announcements %}
      {% endblock body_announcements %}
      {% block body_main %}{% endblock body_main %}
    {% endblock body_content %}
    </div>
```

We define a footer `div`, with a footer breadcrumb, and a hook for anything our company's lawyers asked us to put:

```
    <div id="footer">
    {% block body_footer %}
      {% block body_footer_breadcrumb %}
        {{ page.breadcrumb }}
      {% endblock body_footer_breadcrumb %}
      {% block body_footer_legal %}
      {% endblock body_footer_legal %}
    {% endblock body_footer %}
```

Now we close that `div`, the body, and the body block:

```
    </div>
    </body>{% endblock body %}
```

We add a footer, with JavaScript blocks, again at the site/section/page level of hooks:

```
{% block footer %}
  {% block footer_javascript %}
      {% block footer_javascript_site %}
        <script language="JavaScript" type="text/javascript"
                src="/static/js/jquery.js"></script>
      {% endblock footer_javascript_site %}
      {% block footer_javascript_section %}
      {% endblock footer_javascript_section %}
      {% block footer_javascript_page %}
```

```
            {% endblock footer_javascript_page %}
        {% endblock footer_javascript %}
    {% endblock footer %}
</html>
```

And that's it.

You can make as many layers of templates as you want. One suggested approach is to make three layers: one base template for your entire site, then more specific templates for sections of your site (whatever they may be), and then individual templates for end use. The template given is a base template, and it provides hooks as narrow as a specific meta tag or as broad as the main content area.

For our extended example, if it is named `\base.html`, then we can create another template, `russian.html`, which will declare its content to be in the Russian language. (Ordinarily one would do more interesting things in overriding a template than merely replacing tags, but for illustration purposes we will do that, and only that:

```
{% extends "base.html" %}
{% block html_tag %}<html xmlns="http://www.w3.org/1999/xhtml" xml:
lang="ru-RU" lang="ru-RU">{% endblock html_tag %}
{% block head_meta_contentlanguage %}<meta http-equiv="Content-
Language" value="ru-RU" />{% endblock head_meta_contentlanguage %}
```

These block overrides may occur anywhere; while the `{% extends "base.html" %}` tag must be placed first, the two tags may be swapped. It would work just as well to create a language-agnostic `base.html` with only an empty hook:

```
{% block head_meta_contentlanguage %}
{% endblock head_meta_contentlanguage %}
```

and a default HTML tag of:

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

And then create `english.html` and `russian.html`, or `en-US.html` and `ru-RU.html`, as base templates for those languages.

However, there are many other tags than those used for blocks and overriding. We will just barely sample them below, looking at how to display variables, and then other tags.

There are other tags that look like {% ... %}, but I would comment briefly on the variable tags such as {{ page.title }}. The Django templating language uses dotted references, but not in exactly the same way as Python's dotted references. Where Python requires a couple of different things to get values, dotted references provide one-stop shopping in Django's templating. When a reference to {{ page.title }} occurs, it will display page[u'title'] if page[u'title'] is available. If not, it will display page.title if page.title is available as an attribute, and if there is no such attribute, it will display page.title() if page.title() is available, and failing that, if the reference is a non-negative integer like 2, it will display page[2] if page.2 is requested. If all of these fail, then Django defaults to the empty string because it's not acceptable for a professional site to crash because a programming error has the template asking for something that is not available. You can override the empty string by setting TEMPLATE_STRING_IF_INVALID in your settings.py file, and *in development* it may make sense to set TEMPLATE_STRING_IF_INVALID to something like LOOKUP FAILED, but you will want to set it back to the empty string for deployment in any production environment.

At this point, while there are explicit hooks to pull in multiple JavaScript and CSS files, the preferred practice, per Steve Souders's ground rules for client-side optimizations for high performance websites, is: for each page to load initially, you should have one HTML/XHTML page, one CSS file included at the top, and one JavaScript file included at the bottom. The hooks are more flexible than that, but this is intended more as "development leeway" than what the tightened final product should be.

If-then, if-then-else statements, and for loops are straightforward, and else clauses are optional:

```
{% if results %}
  <ul>
    {% for result in results %}
      <li>{% result.title %}</li>
    {% endfor %}
  </ul>
{% else %}
  <p>There were no results.</p>
{% endif %}
```

There are a number of convenience features and minor variations available; these are several of the major features.

Setting JavaScript and other static content in place

For the development server, putting static content, including images, CSS, and static content, is straightforward. For production use, the recommended best practice is to use a different implementation, and Django users are advised to use a separate server if possible, optimized for serving static media, such as a *stripped-down* build of Apache, or nginx. However, for development use, the following steps will serve up static content:

1. Create a directory named `static` within your project. (Note that other names may be used, but do not use `media`, as that can collide with administrative tools.)
2. Edit the `settings.py` file, and add the following at the top, after `import os`:

```
DIRNAME = os.path.abspath(os.path.dirname(__file__))
```

3. Change the settings of `MEDIA_ROOT` and `MEDIA_URL`:

```
MEDIA_ROOT = os.path.join(DIRNAME, 'static/')
…
MEDIA_URL = '/static/'
```

4. At the end of the `settings.py` file, add the following:

```
if settings.DEBUG:
  urlpatterns += patterns('django.views.static',
    (r'^%s(?P<path>.*)$' % (settings.MEDIA_URL[1:],),  'serve', {
      'document_root': settings.MEDIA_ROOT,
      'show_indexes': True }),)
```

This will turn off static media service when `DEBUG` is turned off, so that this code does not need to be changed when your site is deployed live, but the subdirectory `static` within your project should now serve up static content, like a very simplified Apache. We suggest that you create three subdirectories of `static`: `static/css`, `static/images`, and `static/js`, for serving up CSS, image, and JavaScript static content.

Summary

Guido van Rossum, the creator of the Python programming language, for one project asked about different Python frameworks and chose the Django templating engine for his purposes (`http://www.artima.com/weblogs/viewpost.jsp?thread=146606`). This chapter has provided an overview of Ajax and then provided a kickstart introduction to the Django templating engine. There's more to Django than its templating engine, but this should be enough to start exploring and playing.

In learning a new technology, a crucial threshold has been passed when there is enough of a critical mass of things you can do with a technology to begin tinkering, and taking one step often invites the question: "What can we do to take this one step further?". In this chapter, we have provided a kickstart to begin working with the Django templating engine.

In this chapter, we have looked at the idea of Pythonic problem solving, discussed Django and jQuery in relation to Pythonic problem solving, and discussed Ajax as not a single technology, but an *overlaid function* or technique that is overlaid on top of existing technologies. We have taken an overview of what the "usual suspect" technologies are for Ajax; given a kickstart to the Django templating engine, introducing some of its beauty and power; and addressed a minor but important detail: putting static content in place for Django's development server.

This is meant to serve as a point of departure for further discussion of jQuery Ajax in the next chapter, and building our sample application. Interested readers who want to know more of what they can do can read the official documentation at `http://docs.djangoproject.com/en/dev/topics/templates/`.

In the next chapter, we will push further and aim for a critical mass of things we can do. We will explore jQuery, the most common JavaScript library, and begin to see how we can use it to reach the point of tinkering, of having something that works and wondering, "What if we try this?", "What if we try that?", and being able to do it.