1

Web 2.0, Python, and Frameworks

You had me at "Hello."

—Renée Zellweger in *Jerry Maguire*

The authors know you. Or at least they know a few things about you. If you've grabbed this book from your favorite online (or even brick-and-mortar) bookseller, chances are that you're an accomplished Python developer, or you're on your way to becoming one. You may well have created web content using your Python skills, or perhaps you've written your share of web services backed by Python code as a Common Gateway Interface (CGI) script. You may have just heard the term "Web 2.0" or "AJAX," or you may already be an accomplished user of one or many Web 2.0 sites and are wondering how to plug into the action as a developer. You may be aware of web frameworks in other languages, such as Ruby on Rails (also referred to simply as Rails) or Google Widget Toolkit (GWT), and you may even have experimented with one or more them. If you've been unable to embrace any of these other frameworks, it may be because they have one annoying aspect — they're not Python. Actually, if you've worked with other frameworks, the fact that they don't follow the Python philosophy (they're not *Pythonic*) may be the least of your negative impressions.

The good news is that now you don't have to abandon the language that you love to get the (admittedly significant) benefits of a comprehensive framework. The Python-based frameworks covered in the chapters that follow are every bit as capable and fun to use as any other, and it might be argued, perhaps even better in some ways. The Rails language, for example, demands a certain adherence to convention that sometimes gets in the way of a coder's freedom of expression. GWT is essentially a Java-based page-element composer that enables you to bypass writing ECMAScript directly, but lacks the design center for dynamic partial page replacement for which the Python Turbogears framework is becoming renowned.

The Starting Line

The goal of this book is to help you discover a new methodology for designing, coding, testing, and deploying rich applications that reside primarily in the network cloud, rather than primarily on the desktop. This application style is at the heart of so many modern tools and sites that even if you haven't had experience in developing using the model, you have certainly experienced many sites built this way.

The problem with learning an entirely new way to create applications is, as always, where to start. The authors, being developers rather like you, have always believed that one well-worked example is worth hundreds of words of prose. You'll find those in profusion in this book, but it's important to have an understanding of why Web 2.0 is important to you as a developer or entrepreneur and how the frameworks covered will allow you to leverage your Python skills in different and powerful new ways. Thus, that's the starting point for the book, and even though this chapter is short on code, it's long on ideas, so it will be worth your time to at least skim it. To understand why web frameworks were even designed in the first place, you should understand a little about the whole Web 2.0 revolution.

What's Web 2.0?

Skeptics claim that *Web 2.0* is little more than a marketer's buzzword. Mystically inclined types say that, "To know what it means, you have to *know what it means*. Like art or smut, you'll know it when you see it." Pragmatists explain that, "Google Maps *is* Web 2.0. Yahoo! Mail (old style) *is not*." The more curmudgeonly types, such as industry pundit John C. Dvorak, decry the whole meme by saying, "How can you have a Web 2.0 when we can't even explain what 1.0 was?" Finally, others in the *avant garde* claim that "We're already on version 3.0!"

Whatever the characterization, most nondevelopers agree that the phrase is very handy for encapsulating a whole aggregation of cool, modern stuff that's going on. As a developer, though, you can begin at a much different starting line, because you have a far deeper understanding of design mechanisms and implementations than nontechnical types. Web 2.0 begins with understanding the difference between the traditional Rich Client Platform (RCP) model and the emerging Rich Internet Application (RIA) model.

The History of Application Design for Content Creation (In One Easy Dose)

As the title of Neal Stephenson's famous article explains it, "In the Beginning was the Command Line." However, the capsule history in this section begins after the advent of windowed graphical user interfaces (GUIs), and the emergence of powerful operating systems on expensive compute platforms.

The Desktop Era — No Web

You're certainly familiar with the RCP model, even if you know it by another nomenclature, such as *the desktop application model*. Almost every application you've written, used, managed, or deployed over the last three decades on microcomputers adheres to RCP capabilities and limitations. When you think in the mode of an RCP developer, you take for granted that the operating system, data access and storage, and

the user's experience of the application are tightly bound to the desktop, operating in general ignorance of the larger networked world. When you think like an RCP end user, you often feel trapped by your tools in the sense that if you lose the use of them, or if you lose your disk drive, you also lose your data in a most profound way. For example, it will be hard to read your next great novel written in Enormicorp PerfectWord if you don't have Enormicorp PerfectWord.

One effect of the isolation of both the user and the application is that designers and developers of traditional (RCP) applications never considered it desirable to enable the easy sharing of content created by a specific proprietary tool, except through arrangements like OLE or OpenDoc, which open up data sharing on the operating system or on proprietary tools made by the same vendor. Additionally, they never considered building significant community or social aspects into traditional software.

As discussed in the "Design Patterns for Web 2.0" sidebar later in the chapter, community aspects such as user-created content, tagging, and commentary (common in Web 2.0 applications such as YouTube and Wikipedia) would not have made sense in the era of one user executing one application on one computer. User isolation was a condition of life — an application and the content the end user produced existed on a solitary plane.

Sharing and dissemination of data was done by ad hoc means — for example, by attaching it to e-mail. The intended recipient could share in the data only if they bought and supported the correct version of the proprietary tool. This was true during the first couple of decades of modern computing. The gap between the end users' need to create, publish, and share their creations and the ability of pre-Web–era software design to deliver even a modicum of capability in that regard was filled by what was called *desktop publishing* or *DTP*. DTP was King: the big important designs and design ideas were all about putting words, numbers, and static images onto something that looks like a page, and then printing these pages for sharing. Python was an early response to the needs of system administrators and scientific programmers.

And Then the Web (1.0) Happened

Eventually, people began to wonder why they needed to produce paper at all, and thus when the World Wide Web (the Web) sprang into existence, DTP morphed almost overnight into *Web publishing* or WP. Almost overnight, WP became the new King, efficiently producing HTML to drive applications and represent words, numbers, and multiple media types such as still and moving images. When the Web (1.0) came along, agile languages such as Python began to make inroads into areas very unlike the original class of applications written mostly to enable a higher level of scripting. With the emergence of Twisted Python as a networked-applications engine some years ago, a mimetic light bulb lit. People began to think in terms of *frameworks*, and the notion of code that writes code began to circulate. Python, Perl, and PHP developers in particular began to get a sense of empowerment, while puzzling out the reason for all the manufactured buzz over expensive tools being developed by the same proprietary vendors who dominated the RCP and DTP era. Wasn't everything really just an elaboration of a snippet of code that could write a page in response to a URL invocation of a `cgi-bin` script, such as the Python snippet in Listing 1-1 shows?

Listing 1-1: Simple Python HTML page generator

```
print("Content-type: text/html")
page = """
<html>
 <head>
  <title>Hello World Page!</title>
 </head>
 <body>
 <p>Hello World</p>
 </body>
</html>
"""
print page
```

The answer was both yes and no. In the abstract sense, what every website was striving to do was to generate HTML code (initially) to present material as a substitute for the printed page, and then (later) as a substitute for earlier form-based styles that mimicked human interaction. However, there was also an emerging need to create and maintain transactions for a new style of commerce in which the vendor side of the equation was entirely autonomous and could support a variety of free-form interactions. For that, designers had to evolve a style of architecture that clearly separated the manipulation of the underlying model from the management of the user experience, and from the business logic behind the form. This led to the rediscovery of the *model-view-controller* (*MVC*) design center (discussed in more detail in Chapter 3) just in time for the emergence of Web 2.0.

Web 2.0, the Next Generation

The problem with creating applications and tools based on the first set of technologies underlying the Web was that the user interaction model was really rather wretched. Creating a word-processing tool, such as Google Documents, would have been difficult, if not impossible. Consider the capabilities that a typical Web 2.0 word processor exhibits:

- Every keystroke and input is reflected on the screen as soon as it happens.
- Document modifications are silently persisted into the model, which is online and secured from the possibility of loss from the failure of an individual PC.
- The tool must support a reasonable number of formatting capabilities and enable the insertion of structured content such as tables or images.

In short, such a tool must be largely indistinguishable from the earlier generation's RCP equivalent.

The responsiveness that an interactive user needs in such a case could have been partially mimicked in HTML using HTTP in the Web 1.0 era, but because a write-back to the model could have been accomplished only with a synchronous HTTP request, which also would have resulted in a full page refresh, the experience would have been disorienting for the end user. The result certainly wouldn't have looked or performed like a typical RCP. As Figure 1-1 depicts, every outbound call results in a complete page refresh.

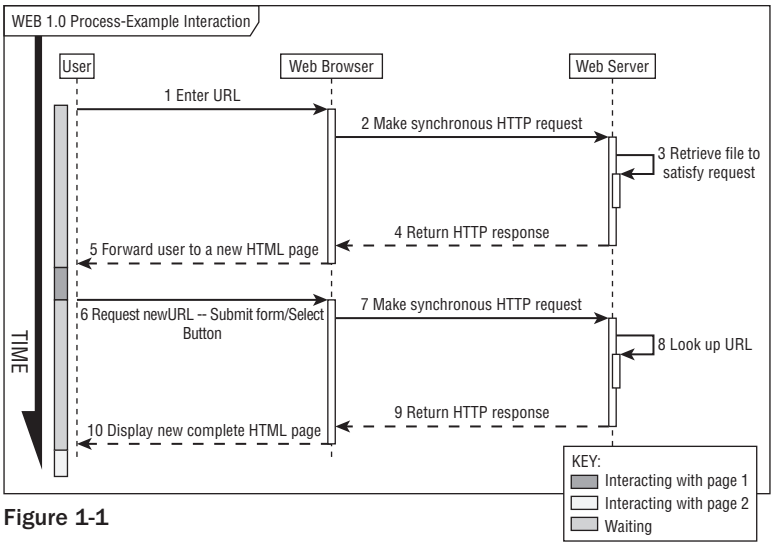

Figure 1-1

What changed the landscape forever was embedding a capability in the browser that enabled it to make an asynchronous call to the server for rendering the web page, and the supporting JavaScript capability to accept the returned data and update a portion of the page (a `<DIV/>` or `<SPAN/>` element of the page) without refreshing the entire page (an idea that Microsoft Internet Explorer 5 initially advanced). This was nothing short of revolutionary and morphed quickly into a realization by developers that the browser could now host applications that mimicked and in some aspects (scalability, robustness, accessibility, and collaboration) improved on the older RCP applications' capabilities.

Thus, the term *Rich Internet Application* (*RIA*) was born, ushering in the Web 2.0 era of tools and sites based on the simple idea that the Internet was becoming a hosting environment, not just for static pages and forms, but also for applications that once were restricted to the desktop.

Applications for such tasks as word processing, photo image manipulation, personal information management — which had been a stronghold for a tiny group of vendors, and were exorbitantly priced with restrictive usage covenants (the infamous and seldom comprehensible *end user license agreement* or *EULA*) — were suddenly left in the dust by browser-based versions that were not only free but consumed no private disk space. All because Web 2.0 applications are designed according to a ubiquitous standard (MVC) and use an asynchronous HTTP request with data returned in XML or *Javascript Object Notation* (*JSON*). The request-and-return path is called an *XMLHttpRequest* or *XHR*, and the technique works in all modern browsers.

Suddenly, word processors, spreadsheets, and calendar management are woven directly into the fabric of the Internet, are executed entirely within the confines of the web browser, and are no longer the private preserve of a specific vendor. What's more, the Web 2.0 design doesn't stop at desktop tool replacements, important as that category is. New classes of applications, such as Base Camp, Flickr, YouTube, and Picasa are useful specifically *because of* their networked context — they wouldn't make much sense as desktop-only applications. In contrast to Figure 1-1, Figure 1-2 illustrates that asynchronous outbound calls may result in a portion of the page being refreshed without a complete browser reload.

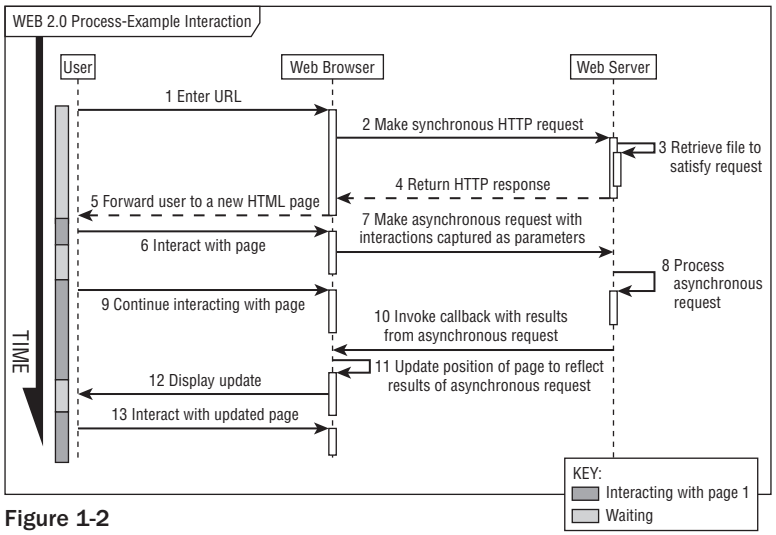

Figure 1-2

This is not to say that there are no problems for designers and developers to solve (yes, we will all still have jobs come the revolution). Even though RIAs usually have several advantages over applications that run on the client, they're not without some limitations. For example, compare an RIA e-mail client like Google Mail (GMail) to a client application like Mozilla Thunderbird. GMail has the advantage of centralized virus scanning, whereas Thunderbird relies on operating-system scanners. On the other hand, when you upload an attachment into an e-mail message, Thunderbird allows you to drag and drop files from any file window into the message, whereas GMail requires you to type in a full file path and name, or use a less convenient file chooser dialog box. Overall, though, these represent browser limitations or operating system limitations, rather than defects in the RIA model itself.

The following table illustrates some of the capabilities you get for free, just by writing to the Web 2.0 specification.

Web 2.0 Capability	Description
Plasticity and flexible recombination	Web 2.0 applications (RIAs) can be remixed into new applications in ways the original designer and developer may never have considered. Because they implement MVC, RIAs expose their server-side invocation syntax to the curious and return their data in a common format (XML or JSON) that any arbitrary process can parse (although there are ways to prevent users from invoking your service, such as requiring private API keys to be passed with the XHR). Overall, this means that your clever server-side logic can be leveraged more widely, gaining you fame, perhaps fortune, and the undying gratitude of your peers.
Invisibility and applications as services	Applications woven into the Web are no longer software — they are services that people take for granted. This is a better thing in the RIA world than it is in the relationship world, because, in the case of your well-written RIA, it may get you the kind of notice that changes your career and lifestyle. Just ask the creators of YouTube what being taken for granted is worth (in their case it was over $1.6 billion).
Ubiquity of access	For almost all Web 2.0 applications, if you have a web browser on a computer somewhere and a broadband connection (both of which are attainable in most of the First World), you can get to the content you have created whether it's the original manuscript for a book or your vacation pictures.
Low switching cost, lower total cost of ownership (TCO)	End users are switching from desktop applications to web-based applications primarily because of the virtues mentioned previously. The cost of switching from the use of tools that shackle the user to services to those that just offer access to their content is actually pretty low. Further, without having to act as an ad hoc system administrator, the cost of using a software capability is lower for the user, and without integrating with a specific operating system, the cost of production is lowered for you, the software developer. In general, Web 2.0 applications eliminate application installation, maintenance, update, and configuration management.
Zero footprint and zero configuration	Although some may argue that a browser is hardly zero footprint, the point is that for the cost of installing and maintaining a modern browser, the end user receives a magic box, from which all manner of applications can spring forth, none of which leaves a footprint on the desktop. The developer (you) receives the gift of a unifying API and development strategy, which is also enabled because of the next point.
Leveraging industry-wide data transport API (HTTP)	This means that the applications you write have worldwide reach.

Table continued on following page

Web 2.0 Capability	Description
Leveraging industry-wide display standards (HTML, Flex, and JavaScript)	This means that you can erase phrases such as *Windows Foundation Classes* (*WFCs*), *Cocoa*, and *Abstract Window Toolkit* (*AWT*) from your memory.
Leveraging a common data format (XML or JSON)	This means that data sharing among applications is straightforward.
Leveraging high-speed networking	This makes running Web 2.0 applications generally as seamless as running desktop applications.
Platform-independent development	Writing to a standard that is above the platform level means that you can develop applications without having to worry about which operating systems, browsers, and so on they will run on (with caveats).

The Role of AJAX in Web 2.0

The ability to make an asynchronous call to a URL and have the data return handled by code within the browser is key to creating the RIA experience for the user. Without *AJAX* (*Asynchronous JavaScript and XML*), browser-resident RIAs would not be feasible for the reasons discussed previously. Additionally, not only can AJAX make asynchronous *external* method calls (out from the browser), but it can also make *internal* method calls (within the browser). The latter capability is the basis for providing changes in visual elements within the RIA, including special effects like growing and shrinking areas, tabbed pages, and drag and drop.

As good and as useful as AJAX is in this regard, the fact is that AJAX must be implemented in JavaScript, and that is the source of a good deal of pain for the developer. Although JavaScript has only recently been taken seriously as a development language, its idiosyncrasies and tendency to be broken in different ways in different browsers is a constant source of consternation to developers and end users alike. Even in a nonbroken world though, it's great to have a level of abstraction sitting atop JavaScript. Fortunately, MochiKit and other libraries are there to help.

> *MochiKit is covered specifically in Chapter 11, but you will find code snippets and explanations of its use in other chapters in this book.*

Leveraging the Power of DSLs

Domain-specific languages (*DSLs*) are limited computer languages designed for a specific class of problem. However, using a special-purpose language that's really good at creating solutions in a specific problem domain is not a brand-new idea. Developers have been creating special syntaxes in languages since way back when, especially in the bygone heyday of early work in Lisp (list processing) and artificial intelligence. In Python and one or two other languages classified as *scripting* languages, creating a special syntax on top of the basic language becomes a highly tractable problem.

This is often called *metaprogramming*, which refers to programming structures that operate on other programming structures. Another way to think of the Python frameworks covered in the following chapters is that they are code that writes code, or even self-modifying code.

As a Python programmer, you may have come across one or two such structures in Python, even if you aren't a master of metaprogramming. The addition of decorators in Python 2.4, which are lines of code preceded by an `@` (at sign), make metaprogramming much less of a mystery and really easy to use. A *decorator* is a function that takes a method as an argument and returns a method (or method-like) object. The Python interpreter generally ignores decorators, but the signs are meaningful to TurboGears, which has over a dozen of them and uses them as part of its controller structure. (Django doesn't use this approach.)

When you mention self-modifying code to most developers, you may evoke a shiver and some whispered words of deprecation, but if it's used correctly and in the right places (as with the frameworks covered in subsequent chapters), such code can lead to some very clean programming, or at least, from your standpoint, very clean-looking code. But don't forget that Python presents broader capabilities to write this type of code than almost any other mainstream language in use today. Long before TurboGears, you could dynamically modify any Python object by adding a new attribute to an instance of an object, or dynamically acquire new methods and add those to an instance.

In contrast to Ruby on Rails, which is almost another language than Ruby, both Python frameworks covered in this book require you to make minimal changes in coding style versus writing normal Python methods. For example, a TurboGears controller method has an `expose` decorator right before the function, which designates the template to be used and returns a Python dictionary filled with the values to be plugged into the template — everything other than that looks just like normal Python code.

You'll find a similar method at work in the Kid templating language, which uses a mixture of standard HTML and some special constructs as a Pythonic control language within the template. Another place you'll see special syntactic constructs at work is in MochiKit, which although implemented atop JavaScript, has method calls and other constructs that look remarkably (and by design) like Python code.

In Chapter 12 you'll see the ultimate pairing of DSLs. That chapter explores the mating of Flash (Adobe's DSL for creating compelling user experiences) with TurboGears. By the end of the book, you will become very familiar and comfortable with the general idea of DSLs as a way to make simple things easy and hard things possible.

Leveraging the Power of TurboGears

By the definition, TurboGears can arguably be called a domain-specific language. What makes its use compelling is that TurboGears is one of a new generation of Web 2.0 frameworks. In this context, the term *framework* means software that enables you to write full-scope software. When people talk about *full scope*, they mean that all aspects of a rich Internet application are addressed, from the user experience, through the controller logic, to the underlying data model.

So why use a framework at all — why not just write dynamic HTML, server logic, and a database handler in straight Python, or Java, or [*insert name of favorite language here*]? The answer to that is the same as it's always been. For every level of abstraction you can move up in software development, you gain an ability to encompass greater complexity.

TurboGears accomplishes the few important goals that matter:

- It enables separation of concerns, which makes maintenance, revision, and the addition of features easier.
- It is excellent in the role of a DSL layer that acts as the binding force for well-crafted Pythonic components at every level:
  - SQLObject handles the model translation from a relational form (from a SQLite or MySQL database) to a Python object form (and back).
  - The Kid templating engine, which can be used standalone at design time as well as in the deployed RIA, is filled in by the TurboGears layer with data from the model. The advantage to running standalone at design time without the backing model or controller logic having been fully developed is that it enables application developers and UI designers to work independently, in parallel, and then converge somewhere along the development path.
  - Within the browser, JavaScript logic is organized through MochiKit, which has the avowed aim to "make JavaScript suck less." Put in more polite language, MochiKit collects a wide array of loosely related JavaScript functions into a single library, and offers a very Pythonic-looking API for method invocation. Chapter 11 covers MochiKit in greater detail.
  - CherryPy creates the web server layer and the control path for transmitting the business logic to the Python code — at least for the development phase. It is likely that for deployment, especially if you use a commercial-grade ISP, you will have to deploy primarily on a web service engine such as Apache.

Leveraging the Power of Django

Django's design center is rooted in the specific domain of content management systems. As such, it's somewhat more focused than TurboGear, which as suggested in the previous section, is much more of a general tool for producing Web 2.0 applications. Thus, where TurboGears is widely focused, Django tends to feel more narrowly focused. Django was created to facilitate creating a daily online newspaper with dynamically generated columns and pages. As you work with both of these frameworks, you will come to appreciate the differences and, more importantly, why the application spaces of interest to different designers forced specific design decisions.

Leveraging the Power of Python

There is probably very little you don't know about Python unless you're new both to Web 2.0 development and to agile languages. But at the risk of telling you things you may already know, there are a few important things that ought to be said of Python as an undercarriage for Web 2.0 frameworks. If you program in Python, you already have experienced some of the attributes that make it a very suitable base.

First of all, Python is an agile, high-level, and interpreted language. For Web 2.0 development, this means that you can write a little, test a little, and deploy a little. This is a perfect quality for iterative development, and is especially perfect for creating the experimental, fast-changing applications in the Web 2.0 space. Applications change quickly here during development and testing, because they are generally driven by the passion of small teams rather than large companies seeking to implement a service-oriented architecture with many moving parts and several months or years to achieve results.

Thus, a language that writes compact, dynamically typed code at a high level of abstraction is a big plus. Additionally, Web 2.0 frameworks need to be able to create specific syntaxes that will speed RIA development without getting in the way. As mentioned in the section "Leveraging the Power of DSLs," a language that enables dynamic self-modification through code that writes code is another important feature, and Python is among the few modern languages that excel in this regard.

Additonally, there are the attributes that you may already cherish. Python is often said to be a solid, *batteries included* (in other words, *complete*) language. Here's why:

- It is solidly supported by a large and passionate community led by a pragmatic, benevolent dictator.
- It is solidly designed to provide functional programming, object-oriented programming (OOP), and metaprogramming.
- It is solidly deployed in countless contexts, from system administration to scientific programming.

Finally, Python's ability to integrate other libraries and components makes it an excellent gap-filling language. It is perfect for automating the many tedious parts of Web 2.0 MVC applications. You should have to write only the parts of your RIA that make it different and compelling, and not all the scaffolding that every RIA must have. Those you will always prefer to delegate to a capable framework. Python helps create the frameworks that make that possible.

Comparing the Frameworks

No parent ever wants to compare their children by saying that Caitlin is better than Sean (they're *both* better, just in different ways) and in this case the same is true. The Django and TurboGears frameworks are each the best at certain things.

Both the Django and TurboGears frameworks implement a strict MVC design center and depend on AJAX within the browser to invoke a server-side controller. Because they are both implemented in Python, they both promote an agile development style. To choose the best one for your needs, consider the differences presented in the following sections.

The Historical Perspective

Both the Django and TurboGears frameworks were motivated by observing that the component parts of specific application implementations were good enough to make them generic. This opened them to the possibility of reusability in other ongoing projects that were quite different from the original. Some of you may recall that Python itself had a similar genesis. TurboGears sprang from an RCP RSS newsreader client. TurboGears tends to be a little more dependent on the momentum of the open source community than Django, simply because of the number of open source components that constitute its moving parts (SQLObject, Kid, and CherryPy). Django originated in response to the need to produce a highly scalable and dynamic online newspaper, one that gets millions of page hits per day.

As a result of the differing design centers, the Django team tends to focus on content management. That's a fancy way of saying that their concern is rooted in the need to create developer methodology

that enables content-based applications to be constructed quickly. The TurboGears team has focused on the RIA space as a whole and has created a highly pluggable architecture. What this means in a practical sense is that even though there are reasonable default choices for supporting componentry, substitutions may easily be made (such as using SQLite instead of MySQL for model representation, or Cheetah instead of Kid for templating).

Controller Invocation

As suggested by Figure 1-2, a URL address in the TurboGears framework results in a server-side call to a method (whose name is a part of the URL) in a file called `controller.py`. As you will discover when you start adding methods to the controller, the method becomes instantly available. You don't have a recompile step nor do you have to restart a virtual machine (VM) or the web server layer. TurboGears provides instant developer gratification.

In Ruby on Rails, there's a lot of chatter about favoring convention over configuration. Django takes the opposite approach because it uses a mapping file to map URLs to the controller code. This strategy decouples the URL path composition from the actual underlying implementation, which in turn enables Django-hosted applications to maximize code reuse and controller deployment.

JavaScript

Much as Prototype is the preferred JavaScript and AJAX library implementation in Rails, TurboGears uses MochiKit to create some very useful binding capabilities for form and effects handling. Django doesn't really have a demonstrated preference for a JavaScript library nor even a loose binding to one in particular. As you'll see demonstrated later in the book, where a project using TurboGears and Flex is shown, there are no real limits to how the user experience or the asynchronous call-and-response system is handled.

Licensing

Licensing is always a critical deployment issue. It's important to know whether you can deploy your creation unencumbered before setting out to build it in the first place. We (the authors of this book) both contribute to and strongly advocate the use of open source. Here are some guidelines to help you do the same:

- Because TurboGears is a composition layer sitting atop other frameworks, it can't be considered independent of its moving parts. Kid, the default templating engine, is licensed under the MIT License (`www.opensource.org/licenses/mit-license.php`), which supports full transitivity with respect to derived works. This means that, basically, you can use it without worry.
- MochiKit is dual-licensed under the terms of the MIT License, or under the Academic Free License (AFL), which is a very tolerant license with respect to the rights to use, deploy, sublicense, and distribute derivative works. SQLObject, the object-relational mapping (ORM) layer, is protected by the Lesser General Public License (LGPL), which is a bit more restrictive. If you modify the source of the SQLObject library itself, you'll have to make those changes available to all. Of course, that doesn't mean that your derived works are in jeopardy. You can create and sell them or give them away without concern.
- Django, because it was created from whole cloth, is under a single open source license (the BSD license). This is very generous with respect to improvements in Django itself and makes no demands on derived works.

Design Patterns for Web 2.0

In the book *Rich Internet Applications: AJAX and Beyond* (Wiley Publishing, 2007), the authors present seven crucial high-level design patterns for Web 2.0 developers. They are repeated here in slightly altered form for your consideration. These are important aspects that create the design center for most of the applications you'll write in this new, highly net-centric paradigm.

Design Pattern 1: Expose (at Least Some of) the Remote API

RIA developers should create content in such a way that mediating applications which other developers or end users might dream up can interact with it, display it, or repurpose its content. User-level content repurposing is extremely common in the Web 2.0 era in the form of *mashups*. Mashups and remixes occur when new applications (originally unintended) are constructed from data and services originally intended for some other purpose.

Design Pattern 2: Use a Common Data Format

A different (but similar) pattern suggests that RIA developers create data models underlying the service-based applications, so that they or another developer can leverage them. No matter what format the back-end server uses to store the data model (regardless of whether it's a SQLite relational database, an NFS-mounted, flat-file system, or an XML data store), the output to the user view should be expressed as something that any number of readers or parsers can handle.

Design Pattern 3: The Browser Creates the User Experience

In most commercially available browsers, application capabilities are limited only by the stability, extensibility, and support for JavaScript. You, the developer, can deal with this limitation because, as a trade-off, it's a lot better than the tall stack of entry limitations and barriers that drove you crazy during the previous software epoch. If you need visual sophistication beyond the capability of the modern browser, you can use other browser-pluggable alternatives, such as Adobe Flash (and write code in Python on the server side and Flex on the browser side).

Design Pattern 4: Applications Are Always Services

Although this was already mentioned briefly, it's a sufficiently important design pattern to bear repetition. RIAs are part of a service-oriented architecture (SOA) in which applications are services, created from scratch or as artifacts from recombining other application facets. Subject to limitations (such as the availability of some other service on which your application may depend), SOA creates the software equivalent of evolutionary adaptation. Remarkably, no protocol stack by a committee of desktop vendors was required to create this pattern. Unlike the UDDI or WDSL stack, the Web 2.0 SOA is simple and almost intuitive.

Design Pattern 5: Enable User Agnosticism

RIAs won't force the user to worry about operating systems or browsers. As long as the browser supports scripting, applications can unobtrusively just do their work. Developers could choose to take advantage of secret handshakes or other backend implementations that assure vendor or OS lockdown (and you can probably think of one browser and one OS vendor that does this), but the downside of making such a devil's

(continued)

bargain is that ultimately you, as a developer, will wind up expending more cycles when, after getting some user traction on a specific platform, you find that your best interests are served by extending your application to every other platform. It's better not to support vendors with their own back-end implementation agenda, and to design from the start to avoid the slippery slope that leads to the trap of relying on vendor- or browser-specific APIs. Thus, it's best if your design steers clear of OS and native library dependencies. That way, you can safely ignore much of the annoying Byzantine detail of writing applications, and specifically the parts that tend to make applications most brittle. Instead, you can concentrate on the logic of what it is that you're trying to accomplish for the user.

Design Pattern 6: The Network Is the Computer

Originally a Sun Microsystems marketing slogan from the 1980s, this bromide goes in and out of vogue. In the 1980s, Java applets exemplified it with the X11/Motif distributed models; in the 1990s, both models ultimately failed, on infrastructural weaknesses. X11 failed because it exacted a huge performance cost on the expensive workstation CPUs of the era, and because its network model couldn't extend beyond the LAN (with the broadband era still a decade or so away). Java applets failed because of their size (long load time) and sluggish performance (again, broadband was a few years away). Today, broadband is near ubiquitous in much of the cyber landscape. In addition, because the current RIA user experience is small and fits comfortably in the memory footprint of the modern browser, failure seems far less likely. Thus, the operating system for this new generation of applications is not a single specific OS at all, but rather the Internet as a whole. Accordingly, you can develop to the separation of concerns afforded by the MVC paradigm, while ignoring (for the most part) the network in the middle of the application. There are still some real concerns about the network in the middle, and given this reality, you do need to code to protect the user experience.

Design Pattern 7: Web 2.0 Is a Cultural Phenomenon

Except for the earliest days of personal computing, when the almost exclusively male Homebrew Computer Club (which gave birth to the first Apple) defined the culture of hackerdom, the PC desktop has never been a cultural phenomenon. In contrast, Web 2.0 has *always* been about the culture. Community is everywhere in the culture of Web 2.0, and end users rally around affinity content. Craigslist, Digg, and Slashdot exist as aggregation points for user commentary. Narrow-focus podcasts have become the preferred alternative to broadcast TV and radio for many; and blogging has become a popular alternative to other publishing mass-distribution formats. Remixing and mashups are enabled by the fact that data models, normally closed or proprietary, are exposed by service APIs, but they also wouldn't exist without end users' passion and sweat equity. Some enterprising developers (such as Ning at `www.ning.com`) have even managed to monetize this culture, offering a framework for creative hobbyists who create social applications based on existing APIs. When you create an application, you should consider the social culture that may arise to embrace your application. You can likely build in collaborative and social capabilities (such as co-editing and social tagging) pretty easily. If there's a downside to having an application exist only "in the cloud," the best upside is that a networked application also obeys the power law of networks: The application's value increases as a log function for the number of users. Most RIAs in the wild have many or all of these characteristics.

> Consider the elder statesman of content-sharing applications, the blog, and its close relative, the wiki. Many blogs approach or exceed print journalism in their reader penetration, and wikis are far more flexible than their power-hungry desktop predecessors (such as Microsoft Office Groove) in supporting multi-user coordination and collaboration.

Summary

In a relatively short span of time, Python developers have seen the Web move from active to interactive to transactive. In the same span, the role of Python has evolved from page generator, to server-side control software, to the basis of exciting and capable frameworks that reduce developer burden while enabling a new class of Rich Internet Applications (RIAs). LAMP (Linux servers running Apache as Web server, backed by a MySQL Relational database, and mediated by Python, Perl, or PHP) has evolved into Web 2.0 and now developers can benefit from making the transition by using the tools and techniques described in this book. It's going to be an exciting and fun ride, so ladies and gentlemen, start your engines.