1

Emergence(y) of the New Web

W. Web knew immediately that something was wrong. He had suffered stomach pangs before, but never like this. Stumbling out of the taxi cab and toward the hospital, he mopped the sweat from his brow and pushed his way through the sidewalk traffic.

Inside, everything was a dizzy blur flowing past him — nurses, patients, a police officer, and several computer technicians hitting a computer monitor and mumbling something about the Internet going down.

''I know, I know!'' Web thought as he struggled past them for the emergency patient entrance.

Luckily for W. Web, this particular hospital uses a triage system, and when you explain to the nurse at the front desk that you are the Internet, you get bumped to the front of the line. A lot is riding on your health.

As Web lay in his hospital gurney, passing other familiar technologies as the nurse pushed him down the hall, he realized that he had made the right decision to stop ignoring the pangs. It was going to be okay.

This book will make you a better web application developer. And if some of the pundits with crystal balls are to be believed, we're all on the path to becoming web application developers. More specifically, this book will make you a better Ruby on Rails developer. It assumes that you have written code using Ruby on Rails and now you are thirsty to understand how to design with Ruby on Rails and how to master the elements of Ruby that make it so successful.

The web is a strange medium for application development. Web applications can't run by themselves, they have little access to a machine's hardware and disk capabilities, and they require a menagerie of client and server software providing them with life support. Despite all this, as you probably already know or are beginning to learn, the Web is a wonderful and exciting place to be developing applications.

Programming for the Web is a blend of art and engineering. Its odd quirks and demands can be harnessed to create applications out of clean, concise, elegant code with minimal waste. This book will show you how.

Programming for the Web is also a task in which good design skills can be the most critical part of a project because of the lack of features such as compilation and type checking found in the desktop world.

Web applications aren't programs; they are ecosystems. For each separate task, a separate language is called upon: SQL for data persistence; Ruby and others for application logic; HTML for UI structure; CSS for UI appearance; and JavaScript for UI logic. Good design skills must extend past the knowledge of each individual area and incorporate the ability to coordinate all areas. On top of all that, the rise of web APIs and RESTful endpoints enable yet another ecosystem of web applications to communicate with each other and exchange services, adding another layer of abstraction that is built upon the ones below.

The Web is here to stay, and its potential will only grow as a development platform. As Internet access approaches utility-like status, as telephone and television did before it, the distinction between your hard drive and ''the cloud'' will blur to the point that the two are functionally indistinguishable. With the exception of maybe games and media editors, applications on the Web will be every bit as powerful as those once deployed on CDs and DVDs, but these new applications will be easier to code, faster to deploy, and will harness the combined intelligence of swarms of users to enrich their experience.

These changes are already taking place. In 2007, the primary process for both the Democratic and Republican parties included a presidential debate with a new twist: Questions for the candidates were asked via YouTube videos submitted by ordinary people through the Web. Web front ends for our banks, stocks, and bills are now considered requirements instead of features. It is no longer surprising to store data that we own, such as our documents and photos, to web applications such as Google Docs and Flickr — space leased for free in exchange for a bit of advertising. The Web is no longer just about fetching documents; instead, it has become a universal communications medium for both humans and software.

If you are here reading this page, then you see these changes taking place. The question that remains is how to best understand and take advantage of this new development medium.

This book aims to be a blend of design and programming. It takes a critical look at what makes the modern Web tick from the standpoint of Ruby on Rails. The chapters touch on a wide range of topics, from REST-based web design to domain-specific languages and behavior-driven development. All these topics represent the cutting edge of thought about web development and will become cornerstones of the web of applications that will flourish over the coming years.

At times throughout the book, the code will be sparse; elsewhere, it will be frequent. In all chapters, long code examples will be avoided in favor of small code examples interwoven with the text to demonstrate an idea. This is a book primarily about concepts, not syntax.

Rails, Art, and the New Web

No development framework understands the new Web better than Ruby on Rails. In a world of general-purpose languages applied to the Web through libraries and Apache modules, Ruby on Rails was the application framework to speak the Web language as its native language. Rails is both a programming framework and a style of development reflected by that framework.

Ruby on Rails embraces the latest thoughts in web design so thoroughly that in many cases it literally forces you to use them. Most other frameworks don't have this option — they have been around so long that entire industries built around them require legacy support. As a newcomer to the scene, Rails is in the unique position of being able to cherry pick both its features and the way that it exposes them to the developer, unifying the framework around these choices. Remember when Apple ditched the floppy drive? It's like that.

This our-way-or-the-highway approach is a bit brazen, but it has a wonderful effect: It yields a remarkably clean framework that makes writing high-quality code in very little time easy. Most of the ''housekeeping'' involved in writing a web application is done for you by the framework, and the rest of your coding tasks are assisted by a host of helper functions and code generators (both code-time and run-time). This means that good Rails developers can spend their time focusing on design-related issues rather than writing code, making sure that each line written is effective and appropriate.

But Ruby on Rails is still a tool, and as with any other tool, it can be misused. Tools that make a point of being simple to use often lull their owners into a false sense of security. The quick learning curve creates the illusion that there isn't anything else to it. Rails, and the Ruby language, are known for being concise, but tidy code doesn't come for free.

Art and Engineering

This book will teach you the finer points of designing and coding in the Ruby on Rails environment — the points that will transform Ruby on Rails from an easy-to-use web tool into a methodology of programming in which every design choice you make is purposeful. Ruby on Rails is a particularly good platform on which to practice this type of purposeful, artful programming because of the way it cuts out the fat in web development to leave only your design remaining.

Software development takes on an inherently artistic quality when the developer truly understands the environment he or she is working in and the tools that are available. Conversely, if you have ever watched a watercolor painter work, you know that art has a lot of engineering in it. Watercolor paintings are intricately designed in advance because each brush stroke can only add to, rather than replace, the color beneath it. Intricate protective masks are applied and layered with the precision of an Intel engineer layering the metal on a silicon wafer.

Ruby on Rails operates with this level of attention to the environment of web development — a level that extends beyond engineering and into art. This book attempts to address the higher-level web application design issues with this same level of attention. With a solid framework underneath and good design skills guiding your programming, your development will become both productive and fun, and these qualities will be reflected in the software that you write.

The New Web

The days of version numbers seemed over when Microsoft Windows suddenly jumped from version 3.11 to 95 overnight, and then advanced 1,905 product releases forward to 2000 in the span of just five years. So what a throwback it seemed when the masses collectively announced that the Web was versioned, too, and it had reached 2.0.

Web 1.0 describes the web as a digital version of the traditional publish-subscribe media model in which a few groups produce content while the majority of users passively consume it. Web 2.0 is a correction

of this imbalance. Web 2.0 applications provide environments in which users can create and publish their own content without having to create and maintain web sites by themselves. Applications such as Blogger, Flickr, Digg, Wikipedia, YouTube, and Facebook turn over the bullhorn to their users, and in doing so have toppled traditional assumptions about the media industry.

Foreshadowed by the prevalence of APIs on the Web today, Web 3.0, as many are calling it, will bring a layer of automated reasoning and programmability to the Web. Semantic search engines will be able to answer questions such as ''which flights will get me to Seattle before lunchtime tomorrow'' instead of simply suggesting web sites associated with the topics ''flights,'' ''Seattle,'' and ''lunch.'' These engines will be able to sift through the Web as a data set, piecing together bits from multiple web sites using semantic markup to align the meaning of the data elements of each. This roadmap for the Web is shown in Figure 1-1.

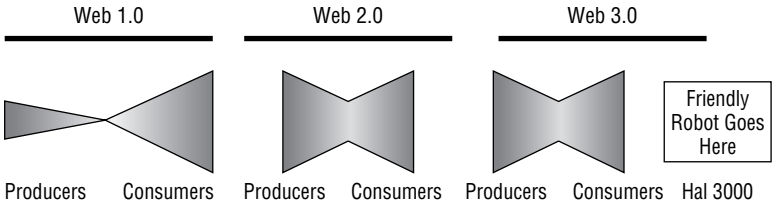

Figure 1-1

Another story is taking place beneath the media headlines and business models, and that is what this book is all about. A true renaissance of web development techniques is making the new capabilities of the Web possible. These advances are breaking long-held assumptions about the way web sites are developed and are introducing a completely new discipline of application development. In the world of web developers, each new ''version'' of the Web reflects a maturing of the art of web development as a discipline.

On the client side, Web 2.0 represented the push for refinement and tidying up of web formats, turning what once was a document markup language into a full-blown client-server application platform. This new platform was made possible because the web development community chose to place a high value on excellence in coding. XHTML and CSS validation icons were displayed as badges of honor at the bottoms of web sites, and the tutorials filling the Web focused on getting rid of the endless `TABLE` tags that clogged auto-generated HTML and on moving instead to simple, hand-crafted XHTML designs decorated entirely via CSS.

On the server side, the changes included new ideas about the ways frameworks should interact with developers, new interpretations of how URLs represent a web site's capabilities, and the incorporation of traditional application programming techniques into web development. In the chapters ahead, you will see how Ruby on Rails is at the forefront of many of these changes and how to incorporate them into your own development.

As the technologies of the Semantic Web are refined, Web 3.0 will be made possible by the introduction of resource-oriented development techniques in web development. The REST development style in Chapter 6 will teach you resource-oriented web design, which is the first step in this process. REST-based web design paves the way for formal modeling and reasoning systems to be directly integrated into our web applications, combining modern web design with the Semantic Web vision of an interlinking web of data and logic. So what is the ''New Web''? The New Web isn't one particular set of technologies or content

models. It is the continued evolution of the art and challenge of web design toward more capability and richer user experience. Web development has come a long way since the early days of CGI scripts and Mosaic, and it is here to stay as the medium through which a new environment of network applications will be born.

The Truth about Web Applications

Unfortunately, these exciting developments on the Web have a catch, and that catch is vitally important to anyone who wants to design good web applications today. The truth is, the Web was not originally designed to be an application platform in the way we are currently using it. As is the spreadsheet, the web is a victim of its own success. It proved to be such a fundamentally simple but flexible tool that its users bent and pried it into increasingly complex roles over the years. Today, we have full-featured web applications displacing the roles of the traditional media and software industry, but these applications are built on top of an architecture that has grown organically over 15 years of development. Web applications are the unexpected child of HTTP and HTML.

This history means that today's applications are still framed and bound by many of the design choices made in the early 1990s when the Web was solely a document publication system. Because we've all been caught up right along in the momentum, many of the oddities created by this fact never received much attention from the scripting community until Ruby on Rails came along; they just seemed to be natural characteristics of the web development environment.

Understanding the unexpected evolution of web applications is essential to understanding how to design a good one. Much of the cutting edge of web design is centered on the idea of returning to the roots of Berners-Lee's original ideas for certain components of web application design while throwing some of our original assumptions out the window in other areas. So although the rest of this book will explore the new world of Rails-based web application development, the rest of this chapter focuses on the evolution of the web from the eyes of a developer.

Patient History: The World Wide Web

For all its short history, the Web has been about *documents*. When Sir Tim Berners-Lee, then employed at the CERN laboratory in Switzerland, created the Web, he was looking for a way to publish and cross-reference project information throughout his lab.

Those were the Dark Ages of computing, when dragons and wizards roamed the net and no two computer architectures could really interoperate. A universal document format that was both hand editable and included the ideas and functionality coming out of the hypertext community at the time would be an enormous improvement to the then-current way of working. Equally important to Berners-Lee's idea was a browser for this new system, which would manage the download and display of these documents automatically, allowing users to follow links from one document to the next.

Berners-Lee had had experience with hypertext before. He had developed a hypertext editor to manage local webs of documents called Enquire, named after a book he received in his childhood titled *Enquire Within Upon Everything*. In his book *Weaving the Web*, Tim Berners-Lee describes his early vision for the Web: an Enquire-like hypertext editing system that was not bound to a single computer.

> My vision was to somehow combine Enquire's external links with hypertext and the interconnection schemes I had developed for RPC. An Enquire

> program capable of external hypertext links was the difference between imprisonment and freedom, dark and light . . . anyone browsing could instantly add a new node connected by a new link. The system had to have one other fundamental property: It had to be completely decentralized.

To facilitate this architecture, Berners-Lee and his colleague Robert Cailliau developed the two technologies that continue to fuel the Web today:

- **HTML** — A language to structure documents and the links between them
- **HTTP** — A protocol for storing and retrieving documents on servers

Equally important, they left us with a legacy of how to think about the web: The web as a distributed document (resource) repository.

Although today the World Wide Web is a virtual world of interaction on the Internet, back then it consisted of two executable programs that Berners-Lee and Cailliau developed for the NeXT system: `WorldWideWeb`, the client, and `httpd`, the server. The `WorldWideWeb` program was a hypertext document editor with a catch: The documents it loaded were specified using Universal Document Identifiers (UDIs, now called URIs) rather than traditional file paths. These UDIs contained both a network host name and a path on that host to specify a file. This meant that in contrast to normal programs, `WorldWideWeb` allowed users to open, edit, and save hypertext documents that were anywhere on the attached network.

The accompanying `httpd` server was responsible for making documents available for remote viewing and editing with the `WorldWideWeb` client. A later version of the `WorldWideWeb` browser is shown in Figure 1-2 (from `http://www.w3.org/History/1994/WWW/Journals/CACM/`) after support for features such as in-page images had been introduced. (The original version could display images, but only as documents themselves, not inline within a hypertext document.)

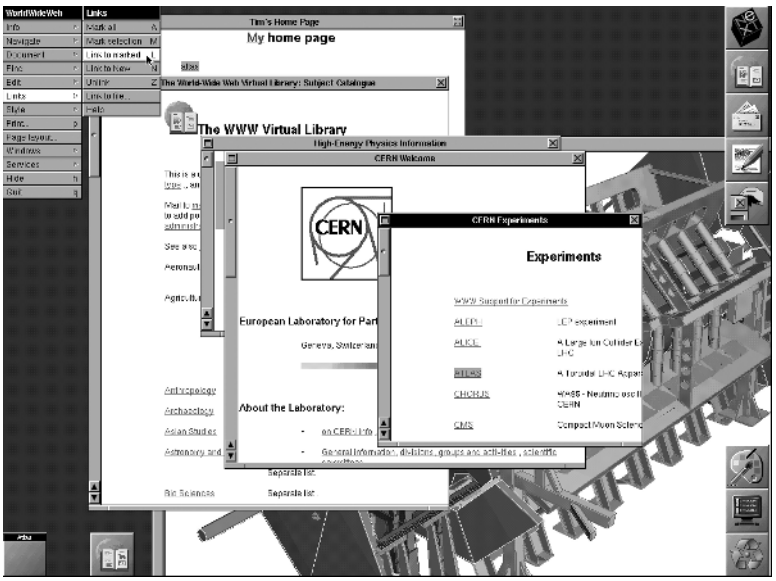

Figure 1-2

The pencil-sketch that Berners-Lee drew in his funding proposal shows a web of concepts, not just documents, interlinked across the network (much like the modern vision of the Semantic Web). But the Web that was immediately needed materialized as a web of published documents. These documents lived on a virtual distributed file system composed of every Internet-accessible computer running the `httpd` server. All that was needed was a URI to specify the server and file path, and any published document in the world could be pulled up, read, edited, and saved back to the `httpd` file server. This basic model is shown in Figure 1-3, a figure that will evolve as the chapter progresses.

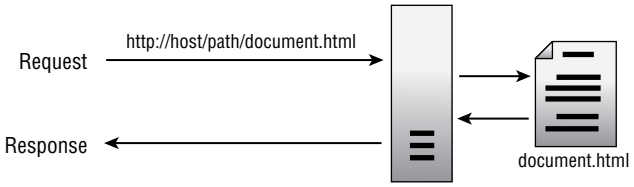

Figure 1-3

If we can characterize the World Wide Web's original behavior as similar to that of a worldwide filesystem, then HTTP is this filesystem's API. The HyperText Transfer Protocol was the mechanism that Berners-Lee designed for performing the operations that one might want to perform on a set of remote files. A working draft of the HTTP specification from 1992 contained 13 method calls in this API, but by the late 1990s, web developers used only two or three with any frequency, and much of these methods' original meanings had disappeared. The following table lists the four primary methods in HTTP commonly used by web developers today — `GET`, `PUT`, `POST`, and `DELETE` — and includes a summary of their meanings as defined by the HTTP 1.1 specification.

Method	Definition
`GET`	The `GET` method means retrieve whatever information is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and now the source text of the process, unless that text happens to be the output of the process.
`POST`	The `POST` method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. `POST` is designed to allow a uniform method to cover the following functions: — Annotating existing resources — Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles — Providing a block of data, such as the result of submitting a form, to a data-handling process — Extending a database through an append operation The actual function performed by the `POST` method is determined by the server and is usually dependent on the Request-URI. The posted entity is subordinate to that URI in the same way that a file is subordinate to a directory containing it, a news article is subordinate to a newsgroup to which it is posted, or a record is subordinate to a database.

Continued

Method	Definition
PUT	The PUT method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity should be considered as a modified version of the one residing on the origin server. If the Request-URI does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI. The fundamental difference between the POST and PUT requests is reflected in the different meaning of the Request-URI. The URI in a POST request identifies the resource that will handle the enclosed entity. That resource might be a data-accepting process, a gateway to some other protocol, or a separate entity that accepts annotations. In contrast, the URI in a PUT request identifies the entity enclosed with the request — the user agent knows what URI is intended and the server must not attempt to apply the request to some other resource.
DELETE	The DELETE method requests that the origin server delete the resource identified by the Request-URI.

From this early draft specification, it is clear that the Web was designed as a distributed document architecture, and all the action was taking place in the HTTP protocol. This Web did not just fetch HTML documents for display, it natively allowed editing and creating, too. New resources could be created with the PUT and POST commands, existing resources could be edited with the PUT command, and resources could be viewed with the GET command. The WorldWideWeb software served as the document viewer and editor for these resources, like a Wiki implemented at the level of HTTP. For example, the following request might be used to change the text of the welcome page to a personal web site:

```
PUT /welcome.html

<HTML>
<HEAD>
<TITLE>Home Page -- Edward Benson's Site</TITLE>
</HEAD>
<BODY>
<H1>Edward Benson's Web Page</H1>
<P>Welcome to my home page! I just modified it with a PUT request!</P>
</BODY>
</HTML>
```

The Web seemed set to provide a distributed document architecture as it spread across the Internet. What Tim Berners-Lee could not have expected was how interpretations of its features would change as the Web moved into the wild and researchers all over the world began making modifications and additions to the web architecture to meet their needs.

From Documents to Interfaces

The first group to dive headfirst into the web was the National Center for Supercomputer Applications at the University of Illinois. Its team included the gang of developers who developed Mosaic, the first web browser for the masses and who later went on to form Netscape. Although Tim Berners-Lee was

responsible for creating and incubating the idea of the World Wide Web, these developers — people such as Marc Andreessen and Eric Bina — are largely responsible for making it a household name.

A great deal of power rests in the hands of whoever writes the interpreter for a computer language because that person or group has unilateral ability to add, remove, or change the way the language is translated into actions by the computer. Microsoft, for instance, is notorious among web developers for single-handedly mutating the way web pages had to be constructed by implementing a flawed rendering engine in early versions of the Internet Explorer browser and then failing to fix the bugs as the versions progressed. The sheer size of the IE market required web developers to treat the exceptions caused by IE's bugs as the new rule. As the web grew in the early 1990s, the Mosaic team had even greater influence by the nature of its role as the keeper of the first mainstream browser.

Two powerful features that the NCSA Mosaic team added to its browser were the ability to display images within HTML documents and the ability to embed forms inside a web page. At the time, these features were controversial in the research community from which the Web came, but their introduction was a result of real-world need rather than research, and they were powerful additions to the web architecture.

Retrospectively, the introduction of `IMG` and `FORM` tag support into NCSA Mosaic was a symbolic event that shaped the future of web development. These two tags set the Web down the path of hosting applications rather than just documents. The `IMG` tag represented the shift of the Web away from an environment consisting only of information and toward an environment in which presentation played a key role. The `FORM` tag represented the shift of the HTML language as a passive medium for information conveyance toward a transactive medium used to facilitate remote database operations. The transactive capabilities that the `FORM` tag enabled also marked the beginning of what would eventually become a complete inversion of control between HTML (the application layer) and HTTP (the transport layer). Although HTTP was once the layer at which information was created and modified, forms allowed HTML to slowly take over and become the place where the real action occurred, leaving HTTP as just the transport mechanism to tunnel data between web forms and the form processor.

The Decline of Semantics

The use of the Web to convey rich document layouts and form-based application interfaces shifted HTML and HTTP away from their original use and semantics. Tags such as `IMG` and `FORM` allowed web documents to be so much more than just informational documents: They could be company home pages, rich advertisements, magazine articles, or user surveys. Similarly to parents of a growing teenager, the original designers of the web could only watch, sometimes with pride and other times with disappointment, its shifting nature as it was adopted and put to use throughout the world.

The Web, as defined by empirical use, broke from its original design and semantics in both of its two major components, HTML and HTTP. Namely,

- HTTP was reduced to only the `GET` and `POST` commands in popular use.
- HTML became a display language.

HTML and the Rise of Pages

What began as a language for describing documents quickly became a language used to describe rich page layouts and rudimentary application interfaces. Driven primarily by commercial interests, but

also by academics excited by a new way to expose an application's functionality to their peers, HTML proved an effective way to produce a particular visual rendering on the client's screen that conveyed aesthetics and branding in addition to structured information. The Web was no longer a repository for just documents; now it hosted ''pages,'' functioning like a digital magazine.

Until stylesheets were developed later in the 1990s, using HTML as a language for UI layout wasn't pretty. The tags in HTML reflect its intent as a document markup language; they are structures of typography rather than layout, such as paragraphs and headings, boldface and italics. But with no choice other than to use HTML, web ''page'' developers began hacking their designs out of its original structures. Of all the tags hacked for the purpose of visual design, the `TABLE` tag became the most important (and abused).

The `TABLE` tag began as an innocuous way to record tabular data. (Makes sense, right?) But a table and its cells also carry with them a certain spatial presence useful for arranging items on a page. Web browsers also supported the ability to specify such properties as border color, height, and width, and so the `TABLE` tag quickly became the staple for constructing user interfaces. Oh, was it painful. With sheer willpower, developers used the flexibility of this tag to coax the Web into a visual medium of its own, but at the expense of readability:

```
...
</td></tr></table></td></tr></table>
<br clear=all>
<table border=0 width=180 cellpadding=1 cellspacing=0 bgcolor=#708090 >
<tr><td>
<table width=100% border=0 cellpadding=4 cellspacing=0 bgcolor=#708090>
<tr><td bgcolor=#ffffff valign=top>
<table border="0" width="100%" cellspacing="0" cellpadding="0">
<TR><TD ALIGN=center CLASS=small BGCOLOR=#FFFFDD>
<table bgcolor="EDEDED" border="0" width="100%" cellspacing="0" cellpadding="0"
hspace="0" vspace="0"><tr>
<td align="center"><a href="Link removed to protect the innocent!">
...
```

This use of HTML as a display language created the web page metaphor that we have today, so, without a doubt, it was an important and exciting building block toward current web applications. But it came at a cost: HTML lost its semantic meaning as a conveyer of documents and became nothing more than an ASCII-based serialization for user interfaces.

Following the dot-com boom of the 1990s, a movement swept across the Web to kick the habit of HTML as a display language and return web design to its roots. This movement, made possible by the spread of CSS to all the major browsers and the development of more evolved versions of the HTML language, is why the `TABLE` tag as a UI structure is now largely a distant memory.

HTTP and the Rise of Forms

Although Berners-Lee's `WorldWideWeb` browser and `httpd` web server were designed to comprise a full-featured hypertext editing system, allowing users to read, create, change, and remove network-hosted HTML documents, NCSA Mosaic and other third-party web browsers supported only viewing HTML documents. With only the ability to view web pages, these web browsers did not have much use for the `POST`, `PUT`, and `DELETE` commands as originally intended. The new `FORM` element created a new

mechanism through which the user could send data to the server, though. By transforming the answers to a fill-out form into a series of key-value pairs, the browser could embed additional user-provided information with the web request.

If the form method was GET, then form data would be encoded and appended directly to the end of the URL in such a way that the server could easily separate the parameters from the document identifier, as follows:

```
GET /search?q=rails&display=100 HTTP/1.0
```

If the form method was POST or PUT, then the encoded form data was sent in lieu of what once would have been HTML content created by the web browser's editor, as follows:

```
POST /search HTTP/1.0

q=rails&display=100
```

With the great array of possible uses of the FORM tag came a realignment of the possibilities of what a web page could represent. Instead of serving merely as a way to publish and modify information, a set of web pages now could together form a transactional interface to some server-side application's capability. This FORM-centric realignment morphed the way HTTP commands were used. Instead of using a full set of resource-oriented operations such as PUT and DELETE, web developers embraced a two-operation mindset: Users were either GETting data or POSTing a form. The four primary HTTP commands shifted in meaning accordingly.

- The DELETE command slowly disappeared from the vocabulary of web developers, a casualty of atrophy.
- The PUT and POST commands ceased to be ways to create and edit web resources and became the mechanisms through which form data was submitted to an application running on the server. Their identical operation made them interchangeable, with POST arising as the dominant choice, arguably because the official definition of the POST command better aligns with form-centric development. Some HTTP servers to this day no longer support PUT by default.
- The GET command ascended to reign over all nonform requests. Whereas formerly the GET command was used only to retrieve a resource without making any changes to data on the server, with the addition of URL-encoded form data, GET requests gained the ability to make changes, too, although it was and is frowned upon to use GET for such operations.

Therefore, although deleting user number 3 should officially be accomplished with a DELETE command:

```
DELETE /users/3
```

today, nobody flinches at the notion of ''posting a delete user'':

```
POST /deleteUser
```

or perhaps even getting one:

```
GET /deleteUser?id=3
```

The form model of programming remains the dominant way to write web applications today. Even the most advanced JavaScript-based applications such as Google Docs are fundamentally organized like a mail-order magazine. Users `GET` read-only pages from this magazine, fill out JavaScript-enhanced forms on the pages, and then `POST` that data back to an application running at the magazine's source. In return, they receive another read-only page whose contents may have been affected by the previous `POST` data. So although web applications such as Wikipedia allow users to create, view, modify, and delete their own interlinking documents in a similar fashion to the original web, they do so at a layer above HTTP commands originally created for these purposes. Users don't ever really edit a Wikipedia *page*; they edit a form containing data about that page and submit it to a script that writes that new data to a database.

In Chapter 6, you will learn about REST-based development, a style of web application development that unites form-based web development with much of the original intent of the HTTP commands. REST represents a whole application architecture defined by web-hosted resources that can be operated upon using HTTP.

Hello, Web Applications

The form-based web development model kicked off the explosion of CGI programs on the Web. Recall that the HTTP `GET` command returns either a document or the results of an executable script at that document's location. With the addition of forms, these remote scripts were able to receive input from the user, creating a whole new range of possibilities. This new architecture is shown in Figure 1-4.

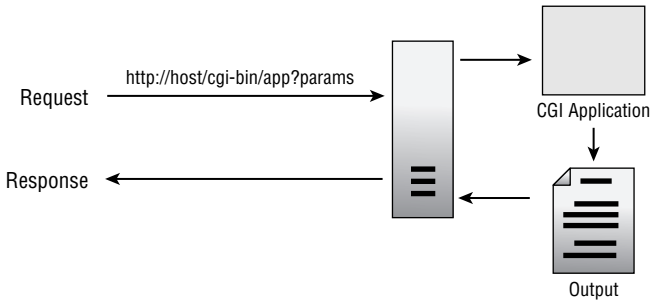

Figure 1-4

In the CGI-driven setup, HTML documents sent to the client represent an interface to a program that resides on the server. These documents contain forms that post to one or more endpoints that the web server knows represents that program rather than a particular file. The nature of the HTTP request is no longer about retrieving a document on the server but instead about sending data to the hosted program and seeing what it has to say in response. The program executed by the web server examines the parameters on the HTTP request, performs some server-side function, and then generates HTML as its output. The following Perl script might be used to process some basic form input, and output a web document, for instance:

```
#!/usr/bin/perl
use CGI qw/:standard :html3/;

my $first = param('first_name') || "unknown";
my $last = param('last_name') || "unknown";
```

```
print header,
   start_html('New User Signup'),
   h1('Thank you for Signing Up!'),
   table({-border=>''},
   caption(strong('Below is a summary of your information')),
   tr({-align=>CENTER,-valign=>TOP},
      [
       th(['Field','Value']),
       td(['First Name', $first]),
       td(['Last Name', $last])
       ]
      )
   ),
   end_html;
```

The output of the CGI script is then sent back to the user as the result of the request, usually as another HTML page containing the last operation's output as well as more forms.

And thus dynamic web sites were born. Built on top of the original HTTP+HTML architecture, a form-based programming model that could provide an interface to server-side software was now possible. CGI scripts were usually written in Perl, but any language could do. The only requirement was that they had to take all their input up front as an HTTP-encoded request and eventually produce their output as HTML.

This is a book about web application design, so the exciting result of CGI is the coding styles that developers used to develop for this new environment. Two styles of coding evolved to support CGI programming and the form-enabled web model, one after the other, which I name here *code-first development* and *document-first development*. Both styles attempted to fix the complications of writing applications that use form-based web interfaces, and the two styles result in very different code.

Code-First Web Development

Code-first development is a programming style that places primary importance on the programming language and secondary importance on the output it produces. The components of a code-first program are filled with functions, classes, and the usual suspects. Any output of the program is assembled using variables, string concatenations, and buffers within the code.

So a Perl program using the CGI.pm module (which is responsible for bringing us such pillars of the web as Slashdot and `Amazon.com`) would render HTML code using helper functions for all of the tags, such as:

```
h1('Thank you for Signing Up!')
```

Or a Java program might use a `StringBuffer`:

```
sb.append("<h2>Thank you for signing up!</h2>");
sb.append("<p>You should be receiving your pickled herring shortly.</p>");
```

The key in both is that the HTML document served by the web request is treated as the output of some program. The web developer doesn't write this document — he writes a program to write it.

The early days of CGI heavily favored this approach because it was the most straightforward way to integrate existing programming languages into the Web. Most programming environments (except for a few, such as LaTeX) were, understandably, program centric rather than document centric. The code-first development style allowed developers to write code in more or less the same way as what they were used to, with the only difference being that the code's output had to be assembled as HTML. These programs would be placed on a filesystem available to the web server, and all that the web server needed to do was execute these programs when a URL referenced them and then send the program's output back to the web user.

The code-first approach offers a number of advantages to the web developer, including:

- It is essentially the same as traditional forms of programming, making it easy to apply well-understood design patterns and testing methodologies.
- It provides complete freedom for the developer, separating the operation of the program completely from the fact that it is being operated in the context of some external service (the web server).

Despite these advantages, its limitations are severe in the context of any nontrivial web application:

- **The HTML produced by code-first programming is not easily maintained.** Anyone who has ever written a Java Servlet without using JSP knows this problem: Scores and scores of concatenated strings assembling fragments of HTML like a person rushing through a store filling his arms with piles of goods. The following code contains a serious HTML error. Can you find it?

```
protected String buildThankYouResponse() {
 StringBuffer sb = new StringBuffer();
 sb.append(beginPage());
 sb.append(title());
 sb.append(beginSidebar());
 sb.append(writeLinkMenu());
 sb.append(beginMainSection());
 sb.append("<h2>Thank you for signing up!</h2>");
 sb.append("<p>You should be receiving your pickled herring shortly.</p>");
 sb.append("<p>Click <a href=\"index.html\">back</a> to return to the book.");
 sb.append(endMainSection());
 sb.append(endPage());
 return sb.toString();
}
```

  The error is just one of many common ones in this type of code. The output of the `beginSidebar()` method is appended to the final response, but an `endSidebar()` method is never called. The incremental, method-calling approach makes committing this kind of mistake easy, potentially leading to nonsensical HTML. Equally as dangerous is the confusion that these method calls add to the process in the first place: Perhaps the `beginSidebar()` function cleans up after itself and does not need an `endSidebar()` counterpart, but without digging into the implementation of each, it is impossible to know.

  HTML is a hierarchical, document-centric language, and it quickly stops making sense when small fragments of it are taken out of their context. Scattered across many lines of code and surrounded by quotes and function calls, it is hard to understand, and it isn't any fun.

- **Code-first programming combines page design with control logic.** When HTML pages are assembled inside the loops and functions of a program, separating the design of the page from the control structures that decide what a user should see becomes impossible. This means that any visual changes to your web page require modifications deep within your application code. Imagine if painting your car required rebuilding the engine. That is what to expect whenever you run across a program that begins like this:

```
beginPage(sb);
if (user.isAdmin()) {
 sb.append("<h1>Administrative Interface</h1>");
 for (Module adminModule : adminModules) {
        adminModule.toHTML(sb);
 }
}
else {
 sb.append("<h1>Welcome, Ordinary User Not Deserving of Cool Admin
Modules!</h1>");
}
```

  For small applications, this limitation may not be a serious problem, but over time, and with scale, it will inevitably become a big one. Page design shouldn't be about programming, and programming shouldn't be about page design. They are separate concerns that are addressed using separate languages, so making one depend on the other only hinders the ability to be effective in either. Web designers with mastery of HTML and CSS but not programming find it difficult to play an active role in web development and maintenance in this environment, for example. Each change they make must be embedded into the flow of a program, so new designs and updates might have to be applied to the site through a programming-minded intermediary. Ideally, web designers could make visual updates to the site with minimal interaction with the program logic that powers its behavior.

- **Code-first programming leads to HTML duplication.** Each code-first program stands on its own, with its own entry point and output, so each is responsible for performing everything required to produce a complete web page. It is possible to organize all these mini-programs so that they share common routines and code, but, in practice, this does not usually happen as well as it could.

  So, seven different CGI scripts all might have a routine that checks to see whether a user is logged in. Or the ubiquitous navigation strip at the top of the page might be duplicated for each page with different styling to signal which page the user is currently on. This type of copy-and-paste programming can be more convenient during development in a code-first approach, but it ultimately makes site maintenance difficult.

Document-First Web Development

Not long after CGI programming swept the Web, a new web development model began to emerge that emphasized the HTML output over the control logic governing it. Document-first programming is a style of programming that places primary focus on the formatted output of the program and secondary focus on the control structures that affect it. Document-first code consists of documents in their output format with embedded code that must be processed before the output is considered complete. Figure 1-5 shows the document-first, or active-document, model graphically.

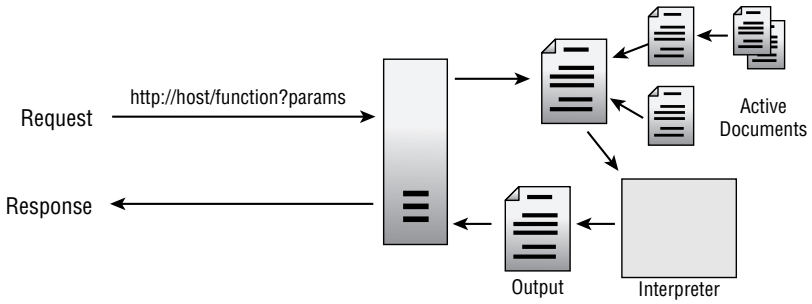

Figure 1-5

For the quintessential example of document-first programming on the Web, look no further than PHP. In 1995, Rasmus Lerdorf wrote a Perl program to help him track accesses to his online résumé. That script turned out to be a bit of overkill, and after two years and a rewrite in C, it became PHP, one of the most successful web scripting languages of all time.

PHP, and the document-first style, presents the programmer with a different environment from the one the first CGI authors were used to. It is an inversion in the primary language used to describe the web application. Instead of writing code that outputs HTML, the programmer writes HTML that contains embedded code. Embedded code is contained within special tags, usually `<% %>` or `<? ?>`.

These documents are stored in a filesystem like any other document that a web user might request, but the server is configured to handle them with special instructions based on their file extensions. Before these documents are returned to the remote user, the web server runs them through a ''hypertext preprocessor'' that processes the document linearly and executes any of the bits of code within it.

So the following fragment might be used to conditionally display an administrative link section to a page based on whether a variable has been set:

```
<? if (user_is_admin) { ?>
<ul class="link_section">
  <li><a href="adduser.php">Add User</a></li>
  <li><a href="resetpassword.php">Reset Password</a></li>
</ul>
<? } ?>
```

Or a list of comments for a blog entry might be shown as follows:

```
<? foreach ($comments as $comment) { ?>
  <div class="comment">
    <h1><?= $comment->title ?></h1>
    <p><?= $comment->contents ?></p>
  </div>
<? } ?>
```

Using the document-first style, web developers can work in their native environment of HTML while still writing the code needed to control decision-making and load data. It preserves the document feeling while still allowing dynamic behavior. The document is its own output — nowhere, as the developer, do

you need to state that a particular portion of text is headed to the remote user. As long as the control code embedded within the document does not prevent a particular region of HTML from being parsed over, it will be appended to the output.

This style of coding is also more web-designer friendly than the code-first style. Despite the presence of embedded code, HTML is the dominant structure, so web designers can work around the control statements and make edits to regions of HTML without having to change any code. Document-first files can be referenced for inclusion to other files, so the heavy bits of programming can be roped off into library files and simply included near the beginning of a page. File-inclusion also allows web designers to avoid repetition, breaking often-used idioms into their own file for reuse across the site. Although the document-first approach presents a much easier way to dynamically build HTML documents, it is not without its own problems from a web *application* perspective:

- **Every file is parsed as potential output.** Document-first programming presents a developer-friendly way to craft HTML output but does so at the cost of providing a straightforward way to write the portions of your application that are pure code. Model objects that encapsulate data in the database and libraries with helper functions must be written inside files that are included at run-time during the preprocessing step, which means that they are parsed as potential contributors to the application's output. You therefore must be on guard against accidentally including any strings in those files that will be appended to the output because such strings might prematurely begin the server's response to the client (before cookies are sent or before a redirect decision is made, for example).

- **Managing the decomposition of documents can be arduous.** There is no centralized entry point in document-first applications; each document that can be addressed via a URL serves as its own entry point. This means that developers must include all dependencies at the top of every web-accessible file. In practice, many of these initializations can be abstracted into a single file that may be included with one line on top of each file. But this still doesn't quite solve the problem, because developers are still left to create their own ''pipeline'' of operations that help fulfill the web request. It would be much nicer if the framework did not require every page to start from scratch and instead provided a single entry point that could handle details such as database connections for you.

  The decomposition problem is made more difficult by the fact that a single document type is used to store every possible construct within the project. Whether you are defining a class, writing a set of functions, performing a login operation, or outputting HTML, the setup of the file looks identical. Where should cookie and session-keeping operations go? What about database management code? Should SQL queries be allowed to occur in the middle of a page definition? How should errors be handled at various stages of the page's parsing? These types of questions are all made difficult to answer because all documents in a document-first system such as PHP are treated the same: They all are potential entry points into the application containing embedded code. Without strong guidelines from the framework about proper decomposition strategies, developers are left to find a solution on their own, and the solution will vary from developer to developer.

- **Document-first programming can result in just as much code juxtaposition as code-first programming.** Data operations such as performing a search, loading a user object, or saving a new object must occur inside code blocks embedded in HTML. Although this requirement doesn't present any technical problems for the developer, it is just as poor of a juxtaposition of concerns as concatenating HTML strings inside control flow. There is no reason that developers should have to cope with an SQL statement embedded in a PHP function embedded in an HTML file. Scale this scenario over hundreds of files, and you have a maintenance nightmare.

The document-first approach of hypertext preprocessing has many advantages over the code-first style for web application programming, but it still leaves developers with much to be desired. As long as the code embedded in document-first files directly pertains to the display of information within the document, it seems an efficient and easy-to-maintain solution. But as soon as control logic, database operations, and other such tasks are thrown into the mix, document-first files can become just as difficult to maintain as code-first files.

Emergency Room

The Web has come a long way. It started as a distributed document repository and quickly became the launching board for a new type of application. Propped on top of the original HTML+HTTP architecture, this new application platform shifted the way the architecture was used so that commands and functionality were embedded in the form data of web requests rather than in the HTTP command conveying the request. This approach enabled web requests to convey any type of data, not just document operations, but it also sent the whole industry of applications programming crashing into a medium whose semantics and programming styles were in a state of flux.

So here we are today, with web application development simultaneously revolutionizing our economy and experiencing an emergency. The revolution is occurring because the web provides such a powerful and democratizing platform on which to create applications. The emergency is occurring because web development methods are still in the process of evolving toward the structure and stability required to take on this enormous new role.

If you peered behind the curtains of many web companies during the early 2000s, I suspect you would see waters churning as a result of these two opposing forces. I worked briefly for a major online commerce site at which you've no doubt shopped, and the tension between these two forces could be seen clearly there. The web application that comprised its entire business appeared flawless to the outside web user, but inside it was chaotic. The web application code had grown organically until it reached gigabytes in size, and it contained so many memory leaks that each production server had to be rebooted at semi-random intervals. The company wanted to patch up this software, but it was too large, too intertwined, and too confusing for anyone to attempt such a feat. The result was a multiyear effort to rewrite the entire codebase from scratch. This web application is part of the revolution taking place, both in the Web's importance and in the need for better web development practices.

As the needs of developers change, certain themes that arise more frequently than others become embedded into new environments as built-in idioms to support those needs. Until recently, web application programming has largely been done with a set of keywords and metaphors developed long before the web became a popular place to program. APIs have cropped up to support web-specific features, but they are no replacement for fundamental changes in the programming environment itself. The growth of web applications requires a new type of programming designed specifically for the needs of the Web. Luckily for you, such environments are now beginning to flourish.

Emergence of the New Web

A new breed of development frameworks has appeared that reflects a true maturation of web development as a discipline all its own. With the charge being led by Ruby on Rails, these frameworks represent the idea that web applications are neither code-first nor document-first, but rather a combination of the two. The code-first approach best addresses entry points, control logic, and data operations, whereas the document-first approach best handles the creation of output to be sent to the web client. These new

frameworks reflect a belief that web programming is heterogeneous in language while homogeneous in process, that a successful web framework should not be general-purpose language with a few helper libraries attached but rather a complete environment whose every design feature assumes operation over the Web. As a result, the Web is built into everything from the directory structure of a project to the tasks that no longer have to be performed by the developer.

Ruby on Rails is on the leading edge of web development techniques, both because of its own originality as a framework and its adoption of the latest technologies and ideas from the web development community. This book focuses on both. Some of the chapters cover features or innovations specific to Ruby and the Ruby on Rails framework, but some cover topics in modern web application development embraced by Rails but not monopolized by it.

This chapter outlined the history of web development to highlight not only how much it has changed over the years but also how many of the ground assumptions of the Web have stayed the same. Understanding the assumptions that created your chosen development environment will help you design web applications that integrate into the web more smoothly. And in later chapters, such as Chapter 6, you will see that in many ways, the very latest in web design is a return to the original semantics of HTTP.

There has never been a more exciting time to be a web developer. The web has broken through its adolescent years as a new medium and is now accepted from the living room to Wall Street. As the Ruby on Rails framework demonstrates, the web is no longer just an output file format but a full application development medium of its own, with its own patterns, abstractions, and vocabulary. The remainder of this book shows you how to take hold of this new application medium and use the latest design abstractions and techniques to bring your web development to the next level.