O'REILLY®

Early Release

RAW & UNEDITED

Github

AMPLIFY YOUR SOFTWARE DEVELOPMENT WITH SOCIAL CODING

Chris Dawson
with Ben Straub

The GitHub Book

Chris Dawson and Ben Straub

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

The GitHub Book

by Chris Dawson and Ben Straub

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Meghan Blanchette
Production Editor: FIX ME!
Copyeditor: FIX ME!
Proofreader: FIX ME!

Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition:

2015-07-28: Early Release Revision 1

See *http://oreilly.com/catalog/errata.csp?isbn=0636920043027* for release details.

ISBN: 063-6-920-04302-7

[?]

Table of Contents

Preface

In the 1930s, The Eagle and Child, a pub in Oxford simmered with creativity as JRR Tolkien and CS Lewis philosophized and created their fantasy worlds. Heian court life in 11th century Kyoto cradled Murasaki Shibu and produced The Tale of Genji, Japan's greatest novel. In the 7th century during the Umayyad Caliphate of modern day Damascus, glittering Arabic palaces provided fertile ground for the creation of a new form of poetry, the ghazal ("love poems") the influence of which still courses through modern Arabic poetry.

Each era of major creative innovation has been backdropped by a unique and plumb space where collaboration and creativity flourished. Today, the collaborative gathering space for some of the world's greatest artists (artists who work as software developers) is a virtual town square, a site called GitHub.

Who You Are

This book should be an interesting source of information for people who have used Git or GitHub and want to "level-up" their skills related to these technologies. People without any experience using GitHub or Git should start with an introductory book on these technologies.

You should have a good familiarity with at least one imperative modern programming language. You don't need to be an expert programmer to read this book, but having some programming experience and familiarity with at least one language is essential.

You should understand the basics of the HTTP protocol. The GitHub team uses a very standard RESTful approach for its API. You should understand the difference between a GET request and POST request and what HTTP status codes mean at the very least.

Familiariy with web APIs is helpful, although this book simultaneously aspires to provide a guide showing how a well thought out, well designed, and well tested web API

creates a foundation for building fun and powerful tools. If you have not used web APIs extensively, but have experience using other types of APIs, this is good company.

What You Will Learn

Much of the book focuses on the technical capabilities exposed by GitHub and the powerful GitHub API. Perhaps you feel constrained by using Git only from within a certain toolset; for example, if you are an Android developer accidentally using Git to manage your app source code and want to unlock Git in other places in your life as a developer, this book provides a wider vista to learn about the power of Git and GitHub. If you have fallen into using Git for your own projects and are now interested in using Git within a larger community, this book can teach you all about the "social coding" style pioneered and dogfooded by the GitHub team. This book provides a stepping stone for software developers who have used other distributed version control systems and are looking for a bridge to using their skills with Git and within a web service like GitHub.

Like any seasoned developer, automation of your tools is important to you, and this book provides examples of mundane tasks that we then convert them into automated and repeatable processes, and we show how to do this using a variety of languages talking to the GitHub API.

If you are unfamiliar with the "command line" this book will give you a firm understanding of how to use it, and we bet you will find great power there. To make this book accessible to everyone, regardless of their editor or operating system, many of the programming samples work within the command line. If you have hated the command line since your father forced you to use it when you were five, this is the perfect book to rekindle a loving relationship with the bash shell.

If you absorb not only the technical facets of using GitHub but also pay attention to the cultural and ideological changes offered behind the tools, you'll very likely see a new way of working in the modern age. We focus on these "meta" viewpoints as we discuss the tools themselves to help you see these extra opportunities.

Almost every chapter has an associated repository hosted on GitHub where you can review the code discussed. Fork away and take these samples into your own projects and tools!

Finally, we help you write testable API backed code. Even the most experienced developers often find that writing tests for their code is a challenge, despite the massive body of literature connecting quality code with tests. Testing can be especially challenging when you are testing something backed by an API; it requires a different level of thinking than is found in strict unit testing. To help you get past this roadblock, whenever possible, this book shows you how to write code which interacts with the GitHub API and is testable.

First Class Languages You Need to Know

There are two languages which are so fundamentally linked to GitHub that you do need to install and use them in order to get the most out of this book.

- Ruby: a simple, readable programming language used heavily by the founders of GitHub.
- JavaScript: the only ubiquitous browser side programming language, its importance has grown to new heights with the introduction of NodeJS, rivaling even the popularity of Ruby on Rails as a server side toolkit for web applications, especially for independent developers.

Your time will not be wasted if you install and play with these two tools. Between them you will have a solid toolset to begin exploration of the GitHub API. Several chapters in this book use Ruby or JavaScript, so putting in some time to learn at least a little bit will make the journey through this book richer for you.

Undoubtedly, many of you picking up this book already have familiarity with Ruby or JavaScript/NodeJS. So, the basics and installation of them are in appendices in the back of the book. The appendices don't cover syntax of these languages; we expect you have experience with other languages as a prerequisite and can read code from any imperative language regardless of the syntax. Later chapters which do discuss a facet of the API go into language details at times, and the code is readable regardless of your familiarity with that particular language. These explanatory appendices discuss the history of these tools within the GitHub story as well as important usage notes like special files and installation options.

Who This Book is Not For

If you are looking for a discussion of the GitHub API that focuses on a single language, you will be disappointed to find that we look at the API through many different languages. We do this to describe the API from not only the way the GitHub team designed it to work, but the aspirational way that client library authors made it work within diverse programming languages and communities. We think there is a lot to learn from this approach, but if you are interested in only a specific language and how it works with the GitHub API, this is not the book for you.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/<catalog page>*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Chris wants to thank his lovely wife, Nicole. I hope that I have added to this book even a tiny bit of the wit and wisdom you provide to me and our family every day. My son Roosevelt's energy continues to inspire me and keep me going even when I am at my limits. To my daughter Charlotte, you are my little smiling Buddha. To my mother, who showed me how to write and, most importantly, why to write, which is something we need more of in the technology world. To Tim O'Brien who invited me into this project, thank you, and I hope we can collaborate again.

Ben would like to thank his wife, Becky, for her ongoing support and (when needed) push from behind. None of this would have happened without you.

CHAPTER 1
Introduction

The GitHub API is extremely comprehensive, permitting access and modification of almost all data and metadata stored or associated with a Git repository. Here is a summary of those sections ordered alphabetically as they are on the GitHub API documentation site (*https://developer.github.com/v3/*):

- Activity: notifications of interesting events in your developer life
- Gists: programmatically create and share code snippets
- Git Data: raw access to Git data over a remote API
- Issues: add and modify issues
- Miscellaneous: whatever does not fit into the general API categorization
- Organizations: access and retrieve organizational membership data
- Pull Requests: a powerful API layer on the popular merge process
- Repositories: modify everything and anything related to repositories
- Search: code driven search within the entire GitHub database
- Users: access user data
- Enterprise: specifics about using the API when using the private corporate GitHub

In addition, though not a part of the API, there are other important technologies you should know about when using GitHub which are not covered in the API documentation:

- Jekyll: hosting blogs and static documentation
- Gollum: wikis tied to a repository
- Hubot: a programmable chat robot used extensively at GitHub

With just one or two exceptions, Each of these sections of the GitHub technology stack are covered in various chapters. The GitHub API documentation is a stellar reference which you will use constantly when writing any application that talks to the API, but the chapters in this book serve a different purpose: these chapters are stories about building applications on top of the technologies provided by GitHub. Within these stories you will learn the tradeoffs and considerations you will face when you use the GitHub API. Chapters in this book often cover multiple pieces of the API when appropriate for the story we are telling. We've generally tried to focus on a major API section and limit exposure to other pieces as much as possible, but most chapters do need to bring in small pieces of more than one section.

- The "cURL" chapter: the chapter you are reading now covers a first look at the API through the command line HTTP client called cURL. We talk a bit about the response format and how to parse it within the command line, and also document authentication.
- The "Gist" chapter: covers the Gist API, as well as command line tools and the Ruby language "Octokit" API client.
- The "Gollum" chapter: explains usage of the Gollum command line tool and associated Ruby library (gem) which is backed by Grit, the C-language bindings for accessing Git repositories. We also document some details of the Git storage format and how it applies to storing large files inside of a Git repository, and show how to use the git command line tools to play with this information.
- The "Search" chapter: we build a GUI search app using Python.
- The "Commit Status" chapter: our final chapter documents a relatively new part of the API which documents the interactions between third party tools and your code. This chapter builds an application using C# and the Nancy .NET GitHub API libraries.
- The "Jekyll" chapter: if you push a specifically organized repository into GitHub, GitHub will host a fully featured blog, equivalent in most ways to a Wordpress site (well, except for the complexity part). This chapter documents how to format your repository, how to use Markdown within Jekyll, how to use programmatic looping constructs provided by Liquid Templates, and then shows how to import an entire web site from the Internet Archive into the Jekyll format using Ruby. We show how to respectfully spider a site using caching, a valuable technique when using APIs or third party public information.
- The "Android" chapter: in this chapter we create a mobile application targeting the Android OS. Our application reads and writes information into a Jekyll repository from the Git Data section of the API. We show how to create user interface tests for Android which verify GitHub API responses using the Calabash UI testing tool.

- The "JavaScript" chapter: did you know you can host an entire "single page application" on GitHub? We show how you can build an application backed by a database called GitHub using the JavaScript language. Importanly, we show how you can write a testable JavaScript application that mocks out the GitHub API when needed.
- The "Hubot" chapter: Hubot is a JavaScript (NodeJS) chat robot enabling technologists to go beyond developer operations ("DevOps") to a new frontier called "ChatOps." The Hubot chapter illustrates using the Activities and Pull Requests section of the API. In addition we show how you can simulate GitHub notifications and how to write testable Hubot extensions (which is often a challenge when writing JavaScript code).

We don't cover the organization API: this is a small facet of the API with only the ability to list organizations and modify metadata about your organization; once you have used other parts of the API this nook of the API will be very intuitive.

We also don't cover the users section of the API. While you might expect it to be an important part of the API, the users API is really nothing more than an endpoint to list information about users, add or remove SSH keys, adjust email addresses and modify your list of followers.

There is not a specific chapter on issues. Historically GitHub used to group issues and pull requests into the same API section, but with the growing importance of pull requests they have separated them in the API documentation. In fact, they are still internally stored in the same database and pull requests are, at least for now, just another type of issue. The Hubot chapter documents using pull requests and is a good reference for issues in that way.

The enterprise API works almost exactly the same as the GitHub.com site API. We don't have a chapter telling a story about the enterprise API, but we do provide an appendix which provides a few notes about how to use it with a few API client libraries.

With these chapters we cover the entire API and hope to give you an inside look into the inner workings of the brain of a developer building on top of the GitHub API.

As you might have noticed, this book will take you on an exploration of several different language clients for the GitHub API. Along the way, we'll point out the different idioms and methodologies inherent to those client libraries and shed light on the darker corners of the GitHub API. Don't be alarmed if you thumb through the chapters and see a language which you don't know at all: each chapter is designed so that you can follow along without intimacy to the language or toolkit. You will get the most value if you install the language and associated tools, but the story behind the projects we will build will be interesting even if you don't actually type a line of code from the chapter.

Enough of the theoretical: let's jump into using the API with the powerful cURL tool.

cURL: a starting point for API exploration

There will be times when you want to quickly access information from the API without writing a formal program. Or, when you want to quickly get access to the raw HTTP request headers and content. Or, where you might even question the implementation of a client library and need confirmation it is doing the right thing from another vantage point. In these situations, cURL, a simple command line HTTP tool, is the perfect fit. cURL, like the best unix tools, is a small program with a very specific and purposefully limited set of features for accessing HTTP servers.

cURL gives you the most naked vantage point you will find for the GitHub API (barring using a network traffic analysis tool). cURL, like the HTTP protocol which it speaks intimately, is stateless, meaning it is challenging to use cURL to hit a service and then use the results with a secondary request. We will use cURL in a later chapter within a shell script that explores solutions to this problem, but note that cURL works best with one-off requests.

If you are running these examples on a Linux box, you should be able use your native package management tool to install cURL - either a "sudo yum install curl*" on a RedHat variant or "sudo apt-get install curl" on an Ubuntu (or Debian) system. If you are on any recent version of OSX, you already have cURL installed, but it you can't find it, take a look at the HomeBrew project (*http://brew.sh/*) or MacPorts project (*http://www.macports.org/*). If you are running on Windows or another operating system, you best bet is to download cURL from the cURL web site here: *http://curl.haxx.se/download.html*

Let's make a request. We'll start with the most basic GitHub API endpoint found at *https://api.github.com*.

```
$ curl https://api.github.com
{
  "current_user_url": "https://api.github.com/user",
  "current_user_authorizations_html_url":
  "https://github.com/settings/connections/applications{/client_id}",
  "authorizations_url": "https://api.github.com/authorizations",
  "code_search_url":
  "https://api.github.com/search/code?q={query}{&page,per_page,sort,order}",
  "emails_url": "https://api.github.com/user/emails",
  "emojis_url": "https://api.github.com/emojis",
  ...
}
```

We've abbreviated the response to make it more readable. A few salient things to notice: there are a lot of URLs pointing to secondary information, parameters are included in the URLs, and the response format is JSON.

What can we learn from this API response?

Breadcrumbs to Successive API Paths

The GitHub API is a Hypermedia API. Though a discussion on what constitutes hypermedia deserves an entire book of its own (Check out O'Reilly's "Hypermedia APIs with HTML5 and Node"), you can absorb much of what makes hypermedia interesting by just looking at a response. First, you can see from the API response above that by making a request to the API, you actually get back a map of how you should make additional responses. Not all clients use this information, of course, but one goal behind Hypermedia APIs is that clients can dynamically adjust their endpoints without recoding the client code. In other words, the API should be able to adjust its map, and then clients will adjust themselves, but you as the application developer using the client libraries will not need to understand or even be aware of the changes. If the thought of GitHub changing an API because clients **should** be written to handle new endpoints automatically sounds worriesome, don't fret too much: GitHub is very dilligent about maintaining and supporting their API in a way that most companies would do well to emulate. But, you should know that you can rely on having a API reference inside the API itself, rather than hosted externally in documentation which very easily could turn out to be out of date with the API itself.

This map includes not just URLs, but also information about how to provide parameters to the URLs. For example, the `code_search_url` key references a URL which obviously allows you to search within code on GitHub, but also tells you how to structure the parameters passed to this URL. If you have an intelligent client who can follow this simple format, you could dynamically generate the query without involving a developer who can read API documentation. At least that is the dream that Hypermedia points us to; if you are skeptical, at least know that APIs such as GitHub encode documentation into themselves, and you can bet GitHub has test coverage to prove that this documentation matches the information delivered by the API endpoints. That's a strong guarantee that is sadly missing from many other APIs.

Now let's briefly discuss the format of all GitHub API responses: JSON.

The JavaScript Object Notation (JSON) Format

Every response you get back from the GitHub API will be in the JSON format. JSON is a "lightweight data interchange format" (read more on the JSON.org website). There are other competing and effective formats, such as XML or YAML, but JSON is quickly becoming the defacto standard for web services.

A few of the reasons why JSON is so popular:

- JSON is readable: JSON has a nice balance of human readability when compared to serialization formats like XML.
- JSON can be used within JavaScript with very little modification (and cognitive processing on the part of the programmer). A data format which works equally well on both the client and server side was bound to be victorious, as JSON has been.

You might expect that a site like GitHub, originally built on the Ruby on Rails stack (and some of that code is still live), would support specifying an alternative format like XML, but XML is no longer supported. Long live JSON.

JSON is very straightforward if you have used any other text based interchange format. One note about JSON that is not always obvious or expected to people new to JSON: the format only supports using double-quotes, not single-quotes.

We are using a command line tool, cURL, to retrieve data from the API. It would be handy to have a simple command line tool that also processes that JSON. Let's talk about one such tool next.

Parsing JSON from the Command Line

JSON is a text format, so you could use any command line text processing tool, such as the venerable AWK, to process JSON responses. There is one fantastic JSON specific parsing tool which complements cURL that is worth knowing: "jq". If you pipe JSON content (using the | character for most shells) into jq, you can then easily extract pieces of the JSON using "filters."

jq can be installed from source, using package managers like `brew` or `apt-get`, and there are binaries on the downloads page for OSX, Linux, Windows and Solaris.

Going deeper, in the prior example, let's pull out something interesting from the API map that we receive when we access api.github.com.

```
$ curl https://api.github.com | jq '.current_user_url'
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  2004  100  2004    0     0   4496      0 --:--:-- --:--:-- --:--:--  4493
"https://api.github.com/user"
```

What just happened? The `jq` tool parsed the JSON, and using the `.current_user_url` filter, it retrieved content from the JSON response. If you look at the response again, you'll notice it has key/value pairs inside an associative array. It uses the `current_user_url` as a key into that associative array and prints out the value there.

You also will notice that curl printed out transfer time information (to standard error, so jq did not see it). If we want to restrict that information and clean up the request we should use the `-s` switch, which runs cURL in "silent" mode.

It should be easy to understand how the jq filter is applied to the response JSON. For a more complicated request (for example, we might want to obtain a list of public repositories for a user), we can see the pattern for the jq pattern parameter emerging. Let's get a more complicated set of information, a user's list of repositories and see how we can extract information from the response using jq.

```
$ curl -s https://api.github.com/users/xrd/repos
[
  {
    "id": 19551182,
    "name": "a-gollum-test",
    "full_name": "xrd/a-gollum-test",
    "owner": {
      "login": "xrd",
      "id": 17064,
      "avatar_url":
      "https://avatars.githubusercontent.com/u/17064?v=3",
     ...
  }
]
$ curl -s https://api.github.com/users/xrd/repos | jq '.[0].owner.id'
17064
```

This response is different structurally: instead of an associative array, we now have an array (multiple items). To get the first one, we specify a numeric index, and then key into the successive associative arrays inside of it to reach the desired content: the owner id.

Jq is a great tool for checking the validity of JSON. As mentioned before, JSON key/values are stored only with double-quotes, not single quotes. You can verify that JSON is valid and satisfies this requirement using jq.

```
$ echo '{ "a" : "b" }' | jq '.'
{
  "a": "b"
}
$ echo "{ 'no' : 'bueno' }" | jq "."
parse error: Invalid numeric literal at line 1, column 7
```

The first JSON we pass into jq works, while the second, because it uses invalid single quote characters, fails with an error. Jq filters are strings passed as arguments, and the shell which provides the string to jq does not care, however, if you use single quotes or double quotes, as you see above. The echo command, if you didn't already know, prints out whatever string you provide to it; when we combine this with the pipe character we can easily provide that string to jq through standard input.

Jq is a powerful tool for quickly retrieving content from an arbitray JSON request. Jq has many other powerful features, documented at stedolan.github.io/jq.

We now know how to retrieve some interesting information from the GitHub API and parse out bits of information from that response, all in a single line. But, there will be times when you incorrectly specify parameters to cURL or the API, and the data is not what you expect. Now we'll learn about how to debug the cURL tool and the API service itself to provide more context when things go wrong.

Debugging Switches for cURL

As mentioned, cURL is a great tool when you are verifying that a response is what you expect it to be. The response body is important, but often you'll want access to the headers as well. cURL makes getting these easy with the `-i` and `-v` switches. The `-i` switch prints out request headers, and the `-v` switch prints out both request and response headers (the `>` character indicates request data, and the `<` character indicates response data).

```
$ curl -i https://api.github.com
HTTP/1.1 200 OK
Server: GitHub.com
Date: Wed, 03 Jun 2015 19:39:03 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 2004
Status: 200 OK
X-RateLimit-Limit: 60
...
{
  "current_user_url": "https://api.github.com/user",
  ...
}
$ curl -v https://api.github.com
* Rebuilt URL to: https://api.github.com/
* Hostname was NOT found in DNS cache
*   Trying 192.30.252.137...
* Connected to api.github.com (192.30.252.137) port 443 (#0)
* successfully set certificate verify locations:
*   CAfile: none
  CApath: /etc/ssl/certs
* SSLv3, TLS handshake, Client hello (1):
* SSLv3, TLS handshake, Server hello (2):
...
* CN=DigiCert SHA2 High Assurance Server CA
*        SSL certificate verify ok.
> GET / HTTP/1.1
> User-Agent: curl/7.35.0
> Host: api.github.com
> Accept: */*
>
< HTTP/1.1 200 OK
```

```
* Server GitHub.com is not blacklisted
...
```

With the `-v` switch you get everything: DNS lookups, information on the SSL chain, and the full request and response information.

Be aware that if you print out headers, a tool like jq will get confused because you are no longer providing it with pure JSON.

This section shows us that there is interesting information not only in the body (the JSON data) but also in the headers. It is important to understand what headers are here and which ones are important. There are a lot of them which the HTTP specification requires, and we can often ignore those, but there are a few that are vital when you start making more than just a few isolated request.

All The Headers and Data

Three headers are present in every GitHub API response which tell you about the GitHub API rate limits. They are X-RateLimit-Limit, X-RateLimit-Remaining, and X-RateLimit-Reset. These limits are explained in detail in "GitHub API Rate Limits" on page 19.

The X-GitHub-Media-Type header contains information that will come in handy when you are starting to retrieve text or blob content from the API. when you make a request to the GitHub API you can specify the format you want to work with by sending an Accept header with your request.

Let's look at the full response from the original request.

```
$ curl -i https://api.github.com/
HTTP/1.1 200 OK
Server: GitHub.com
Date: Sat, 25 Apr 2015 05:36:16 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 2004
Status: 200 OK
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 58
X-RateLimit-Reset: 1429943754
Cache-Control: public, max-age=60, s-maxage=60
ETag: "a5c656a9399ccd6b44e2f9a4291c8289"
Vary: Accept
X-GitHub-Media-Type: github.v3
X-XSS-Protection: 1; mode=block
X-Frame-Options: deny
```

```
Content-Security-Policy: default-src 'none'
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: ETag, Link, X-GitHub-OTP, X-RateLimit-Limit, X-RateLimit-Remaining,
Access-Control-Allow-Origin: *
X-GitHub-Request-Id: C0F1CF9E:567A:9610FCB:553B27D0
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
X-Content-Type-Options: nosniff
Vary: Accept-Encoding
X-Served-By: 13d09b732ebe76f892093130dc088652
{
  "current_user_url": "https://api.github.com/user",
  "current_user_authorizations_html_url":
"https://github.com/settings/connections/applications{/client_id}",
  "authorizations_url": "https://api.github.com/authorizations",
  "code_search_url":
"https://api.github.com/search/code?q={query}{&page,per_page,sort,order}",
  ...
  "notifications_url": "https://api.github.com/notifications",
  "organization_repositories_url":
"https://api.github.com/orgs/{org}/repos{?type,page,per_page,sort}",
  "organization_url": "https://api.github.com/orgs/{org}",
  "public_gists_url": "https://api.github.com/gists/public",
  "rate_limit_url": "https://api.github.com/rate_limit",
  "repository_url": "https://api.github.com/repos/{owner}/{repo}",
  ...
}
```

Using this map, is there anything interesting we can retrieve, perhaps information about GitHub itself? We can use the organizational URL and substitute "github" in the placeholder.

```
$ curl https://api.github.com/orgs/github
{
  "login": "github",
  "id": 9919,
  "url": "https://api.github.com/orgs/github",
  "repos_url": "https://api.github.com/orgs/github/repos",
  "events_url": "https://api.github.com/orgs/github/events",
  "members_url":
"https://api.github.com/orgs/github/members{/member}",
  "public_members_url":
"https://api.github.com/orgs/github/public_members{/member}",
  "avatar_url": "https://avatars.githubusercontent.com/u/9919?v=3",
  "description": "GitHub, the company.",
  "name": "GitHub",
  "company": null,
  "blog": "https://github.com/about",
  "location": "San Francisco, CA",
  "email": "support@github.com",
  "public_repos": 106,
  "public_gists": 0,
  "followers": 0,
```

```
  "following": 0,
  "html_url": "https://github.com/github",
  "created_at": "2008-05-11T04:37:31Z",
  "updated_at": "2015-04-25T05:17:01Z",
  "type": "Organization"
}
```

You can see this tells us the company blog (*https://github.com/about*), that the company is located in San Francisco, and the creation date (which strangely does not match their blog post which states April 10th was their official launch date: *https://github.com/blog/40-we-launched*).

So far all of our requests have been to publicly available information. But, the GitHub APi starts getting really interesting when we authenticate and access private information and publicly inaccessible services, like writing data to GitHub. We'll discuss how and why you want to use authentication next.

Authentication

There are two ways to authenticate when making a request to the GitHub API: username and passwords (HTTP Basic), and oAuth tokens.

Username and Password Authentication

You can access protected content inside GitHub using a username and password combination. Username authentication works by using the HTTP Basic authentication supported by the `-u` flag in curl. HTTP Basic Authentication is synonymous with username and password authentication.

```
$ curl -u xrd https://api.github.com/rate_limit
Enter host password for user 'xrd': xxxxxxxx
{
  "rate": {
    "limit": 5000,
    "remaining": 4995,
    "reset": 1376251941
  }
}
```

This cURL command, authenticates into the GitHub API and then retrieves information about our own specific rate limits for our user account, protected information only available as a logged in user.

Benefits of Username Authentication

Almost any client library you use will support HTTP Basic authentication. All the GitHub API clients we looked at support username and passwords. And, writing your own specific client is easy as this is a core feature of the HTTP standard, so if you use any

standard HTTP library when building your own client, you will be able to access content inside the GitHub API.

Downsides to Username Authentication

There are many reasons why username and password authentication is the wrong way to manage your GitHub API access.

- HTTP Basic is an old protocol which never anticipated the granularity of web services. It is not possible to specify only certain features of a web service if you ask users to authenticate with username/passwords.
- If you use a username and password to access GitHub API content from your cell phone, and then access API content from your laptop, you have no way to block access to one without blocking the other.
- HTTP Basic authentication does not support extensions to the authentication flow. Many modern services now support two-factor authentication and there is no way to inject this into the process without changing the HTTP clients (web browsers, for example) or at least the flow they expect (making the browser repeat the request).

All of these problems are solved (or at least supported) with oAuth flows. For these reasons, there are very few reasons to use username and passwords. If you do need simple and quick access to the GitHub API (and you don't use two factor authentication) then HTTP basic authentication can help you in a small subset of use cases.

oAuth

oAuth is an authentication mechanism where tokens are tied to functionality or clients. In other words, you can specify what features of a service you want to permit an oAuth token to carry with it, and you can issue multiple tokens and tie those to specific clients: a cell phone app, a laptop, a smart watch, or even an Internet of Things toaster. And, importantly, you can revoke tokens without impacting other tokens.

The main downside to oAuth tokens is that they introduce a level of complexity that you may not be familiar with if you have only used HTTP Basic which generally only requires an extra header to the HTTP request, or an extra flag to a client tool like cURL.

oAuth solves the problems described above by linking tokens to scopes (specified subsets of functionality inside a webs service) and issuing as many tokens as you need to multiple clients.

Scopes: specified actions tied to authentication tokens

When you generate an oAuth token, you specify the access rights you require. Don't be confused because we start with HTTP Basic to generate the oAuth token: once you have the token, you no longer need to use HTTP Basic in successive requests. If this token is

properly issued, the oAuth token will have permissions to read and write to public repositories owned by that user.

```
$ curl -u username -d '{"scopes":["public_repo"]}' https://api.github.com/authorizations
{
  "id": 1234567,
  "url": "https://api.github.com/authorizations/1234567",
  "app": {
    "name": "My app",
    "url": "https://developer.github.com/v3/oauth_authorizations/",
    "client_id": "00000000000000000000"
  },
  "token": "abcdef87654321
  ...
}
```

The JSON response, upon success, has a token you can extract and use for applications that need access to the GitHub API.

If you are using two factor authentication, this flow requires additional steps, all of which are documented within the [Link to Come].

To use this token, you specify the token inside an authorization header. It is a little bit early to talk about exactly how to interact with the API, but the syntax in cURL looks like the following. For a full flow, check out the Hubot chapter which shows how to use cURL with an oAuth token.

```
$ curl -H "Authorization: token abcdef87654321" ...
```

Scopes clarify how a service or application will use data inside the GitHub API. This makes it easy to audit how you are using the information if this was a token issued for your own personal use. But, most importantly, this provides valuable clarity and protection for those times when a third party application wants to access your information: you can be assured the application is limited in what data it can access, and you can revoke access easily.

Scope Limitations

There is one major limitation of scopes to be aware of: you cannot do fine-grained access to certain repositories only. If you provide access to any of your private repositories, you are providing access to all repositories.

It is likely that GitHub will change the way scopes work and address some of these issues. The great thing about the way oAuth works is that to support these changes you will simply need to request a new token with the scope modified, but otherwise, the application authentication flow will be unchaged.

Be very careful about the scopes you request when building a service or application. Users are (rightly) paranoid about the data they are handing over to you, and will evaluate your application based on the scopes requested. If they don't think you need that scope, be sure to remove it from the list you provide to GitHub when authorizing and consider escalation to a higher scope after you have developed some trust with your users.

Scope Escalation

You can ask for scope at one point which is very limited, and then later ask for a greater scope. For example, when a user first accesses your application, you could only get the user scope to create a user object inside your service, and only when your application needs repository information for a user, then request to escalate privileges. At this point the user will need to approve or disapprove your request, but asking for everything up front (before you have a relationship with the user) often results in a user abandoning the login.

Simplified oAuth Flow

oAuth has many variants, but GitHub uses oAuth2. oAuth2 specifies a flow where:

- the application requests access
- the service provider (GitHub) requests authentication: username and password usually.
- if two-factor authentication is enabled, ask for the OTP (one time password) code.
- GitHub responds with a token inside a JSON payload
- the application uses the oAuth token to make requests of the API.

A real world flow is described in full in the [Link to Come].

Now let's review an important fundamental when using web services, the importance of the HTTP status code.

Status Codes

The GitHub API uses HTTP status codes to tell you definitive information about how your request was processed. If you are using a basic client like cURL, it will be important to validate the status code before you look at any of the data retrieved. If you are writing your own API client, pay close attention to the status code before anything else.

Success (200 or 201)

If you have worked with any HTTP clients whatsoever, you know what the HTTP status code "200" means success. GitHub will respond with a 200 status code when your request destination URL and associated parameters are correct. If your request creates content on the server, then you will get a 201 status code, indicating a successful creation on the server.

```
$ curl -s -i https://api.github.com | grep Status
Status: 200 OK
```

Naughty JSON (400)

If your payload (the JSON you send to a request) is invalid, the GitHub API will respond with a 400 error, as shown below.

```
$ curl -i -u xrd -d 'yaml: true' -X POST https://api.github.com/gists
Enter host password for user 'xrd':
HTTP/1.1 400 Bad Request
Server: GitHub.com
Date: Thu, 04 Jun 2015 20:33:49 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 148
Status: 400 Bad Request
...

{
  "message": "Problems parsing JSON",
  "documentation_url":
  "https://developer.github.com/v3/oauth_authorizations/#create-a-new-authorization"
}
```

Here we attempt to generate a new gist by using the endpoint described at the Gist API documentation: *https://developer.github.com/v3/gists/#create-a-gist*. We'll detail Gists in more detail in a later chapter. This issue fails because we are not using JSON (this looks it could be YAML, which we will discuss in the Jekyll chapter). The payload is sent using the `-d` switch. GitHub responds with advice on where to find the documentation for the correct format at the `documentation_url` key inside the JSON response. Notice that we use the `-X POST` switch and value to tell cURL to make a POST request to GitHub.

Improper JSON (422)

If any of the fields in your request are invalid, GitHub will respond with a 422 error. Let's attempt to fix the previous request. The documentation indicates the JSON payload should look like this:

```
{
  "description": "the description for this gist",
  "public": true,
```

```
  "files": {
    "file1.txt": {
      "content": "String file contents"
    }
  }
}
```

What happens if the JSON is valid, but the fields are incorrect?

```
curl -i -u chris@burningon.com -d '{ "a" : "b" }' -X POST
https://api.github.com/gists
Enter host password for user 'chris@burningon.com':
HTTP/1.1 422 Unprocessable Entity
...

{
  "message": "Invalid request.\n\n\"files\" wasn't supplied.",
  "documentation_url": "https://developer.github.com/v3"
}
```

There are two important things to note: first, we get a 422 error, which indicates the JSON was valid, but the fields were incorrect. We also get a response which indicates why: we are missing the `files` key inside the request payload.

Successful Creation (201)

Now, let's use valid JSON and see what happens:

```
$ curl -i -u xrd \
-d '{"description":"A","public":true,"files":{"a.txt":{"content":"B"}}} \
https://api.github.com/gists
Enter host password for user 'xrd':
HTTP/1.1 201 Created
...

{
  "url": "https://api.github.com/gists/4a86ed1ca6f289d0f6a4",
  "forks_url":
  "https://api.github.com/gists/4a86ed1ca6f289d0f6a4/forks",
  "commits_url":
  "https://api.github.com/gists/4a86ed1ca6f289d0f6a4/commits",
  "id": "4a86ed1ca6f289d0f6a4",
  "git_pull_url": "https://gist.github.com/4a86ed1ca6f289d0f6a4.git",
  ...
}
```

Success! We created a gist and got a 201 status code indicating things worked properly. To make our command more readable we used the backslash character to allow parameters to span across lines. Also, notice the JSON does not require whitespace which we have completely removed from the string passed to the `-d` switch (in order to save space and make this command a little bit more readable).

Nothing Has Changed (304)

304s are like 200s in that they say to the client: yes, your request succeeded. They give a little bit of extra information, however, in that they tell the client that the data has not changed since the last time the same request was made. This is valuable information if you are concerned about your usage limits (and in most cases you will be). You need to trigger 304s manually by adding conditional headers to your request.

Conditional Requests to Avoid Rate Limitations

If you are querying the GitHub APIs to obtain activity data for a user or a repository, there's a good chance that many of your requests won't return much activity. If you check for new activity once every few minutes, there will be time periods over which no activity has occurred. These requests, these constant polls still use up requests in your rate limit even though there's no new activity to be delivered.

In these cases, you can send conditional HTTP headers If-Modified-Since and If-None-Match to tell GitHub to return an HTTP 304 response code telling you that nothing has been modified. When you send a request with a conditional header and the GitHub API responds with a HTTP 304 response code, this request is not deducted from your rate limit.

The following command listing is an example of passing in the If-Modified-Since HTTP header to the GitHub API. Here we've specified that we're only interested in receiving content if the Twitter Boostrap repositories has been altered after 7:49 PM GMT on Sunday, August 11, 2013. The GitHub API responds with a HTTP 304 response code which also tells us that the last time this repository changed was a minute earlier than our cutoff date.

```
$ curl -i https://api.github.com/repos/twbs/bootstrap \
          -H "If-Modified-Since: Sun, 11 Aug 2013 19:48:59 GMT"
HTTP/1.1 304 Not Modified
Server: GitHub.com
Date: Sun, 11 Aug 2013 20:11:26 GMT
Status: 304 Not Modified
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 46
X-RateLimit-Reset: 1376255215
Cache-Control: public, max-age=60, s-maxage=60
Last-Modified: Sun, 11 Aug 2013 19:48:39 GMT
```

The GitHub API also understands HTTP caching tags. An ETag, or Entity Tag, is an HTTP header that is used to control whether or not content that you have previously cached is the most recent version. Here's how your systems would use ETag:

- Your server requests information from an HTTP server.
- Server returns an ETag header for a version of a content item.

- Your server includes this ETag in all subsequent requests.
  - — If the server has a newer version it returns new content + a new ETag
  - — If the server doesn't have a newer version it returns an HTTP 304

The following command listing demonstrates to commands. The first curl call to the GitHub API generates an ETag value, and the second value passes this ETag value as an If-None-Match header. You'll note that the second response is an HTTP 304 which tells the caller that there is no new content available.

```
$ curl -i https://api.github.com/repos/twbs/bootstrap
HTTP/1.1 200 OK
Cache-Control: public, max-age=60, s-maxage=60
Last-Modified: Sun, 11 Aug 2013 20:25:37 GMT
ETag: "462c74009317cf64560b8e395b9d0cdd"

{
  "id": 2126244,
  "name": "bootstrap",
  "full_name": "twbs/bootstrap",
  ....
}

$ curl -i https://api.github.com/repos/twbs/bootstrap \
          -H 'If-None-Match: "462c74009317cf64560b8e395b9d0cdd"'

HTTP/1.1 304 Not Modified
Status: 304 Not Modified
Cache-Control: public, max-age=60, s-maxage=60
Last-Modified: Sun, 11 Aug 2013 20:25:37 GMT
ETag: "462c74009317cf64560b8e395b9d0cdd"
```

If you are developing an application that needs to make a significant number of requests to the GitHub API over a long period of time, you can use a caching HTTP proxy like Squid to take care of automatically caching content, storing content alongside ETags, and injecting the "If-None-Match" header into GitHub API requests. If you do this, you'll be automating the injection of conditional headers and helping to reduce the overall load on the GitHub API.

Use of conditional request headers is encouraged to conserve resources and make sure that the infrastructure that supports GitHub's API isn't asked to generated content unnecessarily.

You might now be wondering: what are my rate limits and when should I care about them?

GitHub API Rate Limits

GitHub tries to limit the rate at which users can make requests to the API. Anonymous requests, requests that haven't authenticated with either a username/password or OAuth information, are limited to 60 requests an hour. If you are developing a system to integrate with the GitHub API on behalf of users, clearly 60 requests per hour isn't going to be sufficient.

This rate limit is increased to 5000 requests per hour if you are making an authenticated request to the GitHub API, and while this rate is two orders of magnitude larger than the anonymous rate limit, it still presents problems if you intend to use your own GitHub credentials when making requests on behalf of many users.

For this reason, if your web site or service uses the GitHub API to request information from the GitHub API, you should consider using OAuth and make requests to the GitHub API using your user's shared authentication information.

There are actually two rate limits. The "core" rate limit and the "search" rate limit. The rate limits explained in the previous paragraphs were for the core rate limit. For search, requests are limited at 20 requests per minute for authenticated user requests and 5 request per minute for anonymous requests. The assumption here is that search is a more infrastructure intensive request to satisfy and that tighter limits are placed on its usage.

Note that GitHub tracks anonymous requests by IP address. This means that if you are behind a firewall with other users making anonymous requests, all those requests will be grouped together.

Reading Your Rate Limits

Reading your rate limit is straightforward, just make a GET request to /rate_limit. This will return a JSON document which tells you the limit you are subject to, the number of requests you have remaining, and the timestamp (in seconds since 1970). Note that this timestamp has a timezone in Coordinated Universal Time (UTC).

The following command listing uses curl to retrieve the rate limit for an anonymous request. This response is abbreviated to save space in this book, but you'll notice that the quota information is supplied twice: once in the HTTP response headers and again in the JSON response. The rate limit headers are returned with every request to the GitHub API, so there is little need to make a direct call to the /rate_limit API.

```
$ curl https://api.github.com/rate_limit
{
  "resources": {
    "core": {
```

```
      "limit": 60,
      "remaining": 48,
      "reset": 1433398160
    },
    "search": {
      "limit": 10,
      "remaining": 10,
      "reset": 1433395543
    }
  },
  "rate": {
    "limit": 60,
    "remaining": 48,
    "reset": 1433398160
  }
}
```

60 requests over the course of an hour isn't very much, and if you plan on doing anything interesting, you will likely exceed this limit quickly. If you are hitting up against the 60 requests per minute limit, you will likely want to investigate making authenticated requests to the GitHub API. We'll show that when we discuss authenticated requests.

Calls to the Rate Limit API are not deducted from your Rate Limit. Isn't that nice of them?

At this point we have been accessing the GitHub API from a cURL client, and as long as our network permits it, we can do whatever we want. The GitHub API is accessible in other situations as well, like from within a browser context, and certain restrictions apply there, so let's discuss that next.

Accessing Content from the Web

If you are using the GitHub API from a server side program or the command line then you are free to issue any network calls as long as your network permits it. If you are attempting to access the GitHub API from within a browser using JavaScript and the XHR (XmlHttpRequest) object, then you should be aware of limitations imposed by the browser's same-origin policy. In a nutshell, you are not able to access domains from JavaScript using standard XHR requests outside of the domain from which you retrieved the original page. There are two options for getting around this restriction, one clever (JSON-P) and one fully supported but slightly more onerous (CORS).

JSON-P

JSON-P is a browser hack, more or less, that allows retrieval of information from servers outside of the same-origin policy. JSON-P works because <script> tags are not checked against the same-origin policy; in other words, you can specify your page should load scripts from any domain and the browser will permit it. With JSON-P, you load a JavaScript file which resolves to a specially encoded data payload wrapped in a callback function you implement. The GitHub API supports this syntax: you request a script with a parameter on the URL indicating what callback you want the script to execute once loaded.

We can simulate this request in cURL:

```
$ curl https://api.github.com/?callback=myCallback
/**/myCallback({
  "meta": {
    "X-RateLimit-Limit": "60",
    "X-RateLimit-Remaining": "52",
    "X-RateLimit-Reset": "1433461950",
    "Cache-Control": "public, max-age=60, s-maxage=60",
    "Vary": "Accept",
    "ETag": "\"a5c656a9399ccd6b44e2f9a4291c8289\"",
    "X-GitHub-Media-Type": "github.v3",
    "status": 200
  },
  "data": {
    "current_user_url": "https://api.github.com/user",
    "current_user_authorizations_html_url":
    "https://github.com/settings/connections/applications{/client_id}",
    "authorizations_url": "https://api.github.com/authorizations",
    ...
  }
 })
```

If you used the same URL we used above inside a script tag on a web page (`<script src="https://api.github.com/?callback=myCallback" type="text/javascript"></script>`), your browser would load the content displayed above, and then a JavaScript function you defined called `myCallback` would be executed with the data shown. This function could be implemented like this inside your web page.

```
<script>
function myCallback( payload ) {
  if( 200 == payload.status ) {
    document.getElementById("success").innerHTML = payload.data.current_user_url;
  } else {
    document.getElementById("error").innerHTML = "An error occurred";
  }
}
</script>
```

This example demonstrates taking the `current_user_url` from the data inside the payload and putting it into a DIV, one that might look like `<div id="success"></div>`.

Because JSON-P works via <script> tags, only GET requests to the API are supported. If you only need read-only access to the API, JSON-P can fulfill that need in many cases, and it is easy to configure.

If JSON-P seems too limiting or hackish, CORS is a more complicated but official way to access external services from within a web page.

CORS Support

CORS is the "correct" way to access GitHub content from within a browser context. CORS requires that the server be properly configured in advance; the server must be indicate when queried that it allows cross domain requests. If the server effectively says "yes, you can access my content from a different domain" then CORS requests are permitted. The HTML5Rocks website has a great tutorial explaining many details of CORS: *http://www.html5rocks.com/en/tutorials/cors/*.

Because XHR using CORS allows the same type of XHR requests as you get from the same domain origin, you can make requests beyond GET to the GitHub API: POST, DELETE and UPDATE. Between JSON-P and CORS there are options to retrieve content from the GitHub API. The choice is between the simplicity of JSON-P and the power and extra configuration of CORS.

We can prove using cURL that the GitHub API server is responding correctly for CORS request. In this case we only care about the headers, so we use the `-I` switch which tells cURL to make a HEAD request, telling the server not to respond with body content.

```
curl -I https://api.github.com
HTTP/1.1 200 OK
Server: GitHub.com
...
X-Frame-Options: deny
Content-Security-Policy: default-src 'none'
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: ETag, Link, X-GitHub-OTP,
X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset,
X-OAuth-Scopes, X-Accepted-OAuth-Scopes, X-Poll-Interval
Access-Control-Allow-Origin: *
X-GitHub-Request-Id: C0F1CF9E:07AD:3C493B:557107C7
Strict-Transport-Security: max-age=31536000; includeSubdomains;
preload
```

We can see the "Access-Control-Allow-Credentials" header is set to true. It depends on the browser implementation, but some JavaScript host browsers will automatically make a "preflight" request to verify this header is set to true (and that other headers, like the "Access-Control-Allow-Origin" are set correctly and permit requests from that origin

to proceed). Other JavaScript host browsers will need you to make that request. Once the browser has used the headers to confirm that CORS is permitted, you can make XHR requests to the GitHub API domain as you would any other XHR request going into the same domain.

We've covered much of the details of connecting and dissecting the GitHub API, but there are a few other options to know about when using it. One of them is that you can use the GitHub API service to provide rendered content when you need it.

Specifying Response Content Format

When you send a request to the GitHub API, you have some ability to specify the format of the response you expect. For example, if you are requesting content that contains text from a commit's comment thread, you can use the Accept header to ask for the raw markdown or for the HTML this markdown generates. You also have the ability to specify this version of the GitHub API you are using. At this point, you can specify either version 3 or beta of the API.

Retrieving Formatted Content

The Accept header you send with a request can affect the format of text returned by the GitHub API. As an example, let's assume you wanted to read the body of a GitHub Issue. An issue's body is stored in markdown and will be sent back in the request by default. If we wanted to render the response as HTML instead of markdown, we could do this by sending a different accept header, as the following cURL commands demonstrate.

```
$ URL='https://api.github.com/repos/rails/rails/issues/11819'
$ curl -s $URL | jq '.body'
"Hi, \r\n\r\nI have a problem with strong...." # ❶
$ curl -s $URL | jq '.body_html'
null # ❷
$ curl -s $URL \
-H "Accept: application/vnd.github.html+json" | jq '.body_html'
"<p>Hi, </p>\n\n<p>I have a problem with..." # ❸
```

❶ Without specifying an extra header, we get the internal representation of the data, sent as markdown.

❷ Note that if we don't request the HTML representation, we don't see it in the JSON by default.

❸ If we use a customized accept header like in the third instance, then our JSON is populated with a rendered version of the body in HTML.

Besides "raw" and "html" there are two other format options that influence how Markdown content is delivered via the GitHub API. If you specify "text" as a format, the issue body would have been returned as plaintext. If you specify "full" then the content will

be rendered multiple times including the raw Markdown, rendered HTML, and rendered plaintext.

In addition to controlling the format of text content, you can also retrieve GitHub blobs either as raw binary or as a BASE64 encoded text. When retrieving commits, you can also specify that the content be returned either as a diff or as a patch. For more information about these fine-grained controls for formatting, see the GitHub API documentation.

Before you start building a system atop another service's API, it is always wise to understand what, if any, limitations are placed on that API's usage. Aside from the limitations on bandwidth, GitHub's API is also covered by the overall GitHub Terms of Service. You can read these terms of service here: *https://help.github.com/articles/github-terms-of-service*

GitHub Has Amazing API Documentation

The GitHub team has already provided very thorough documentation on their API with examples using cURL. Bookmark this URL: *https://developer.github.com/v3/*. You'll use it often. Do note that this URL is tied, obviously, to the current API "Version 3", so this URL will change when a new version is released.

Summary

In this chapter we learned how to access the GitHub API from the simplest client available: the command line cURL HTTP tool. We also explored the API by looking at the JSON and played with a command line tool (jq) that when paired with cURL gives us the ability to quickly find information in the often large body of data the GitHub API provides. We learned about the different authentication schemes supported by GitHub, and also learned about the possibilities and tradeoffs when accessing the GitHub API from within a browser context.

In the next chapter we will look at Gists and the Gist API. We'll use Ruby to build a Gist display program, and host all source files for the application as a Gist itself.

CHAPTER 2

Gists and the Gist API

GitHub revolutionized software development by responding to a deep desire to share information. But calling it just "sharing" does a disservice to the tools GitHub provides: these tools remove barriers to communication and streamline workflows and these tools also arose at exactly the moment when more and more companies permitted and more and more complementary technologies appeared to allow an emerging remote workforce. Gists service part of this need: they permit intimate code sharing and reuse, refactoring and play in a way not served by heavyweight tools predating it.

Gists are easy to create and the interface is stripped down to the barest level. You add a snippet of code and then share the URL. Gists autodetect the language in most cases and format it correctly. Gists can be used in more powerful ways than might appear at first glance and this chapter will explore other ways to share code and amplify your team.

To create a gist, go to gist.github.com and enter in any textual data. You then choose public or secret access and create the gist. After creating the gist, you receive a shareable URL with the code. If the type of textual data is specified, usually the coding language type, then the code will be formatted in a pretty way for better readability. If you need to share a small bit of code, or write something and discuss it, gists are a great tool.

There are other services that do this: pastebin was the first, and there are many others that offer variances on code sharing. But gists by GitHub are not simply a pasting service. Gists are first class repositories, forkable, editable and expansive. We'll go over the basics of what gists are, and how to create them, and then show how they allow you to share code that is also a live application.

Gists are repositories

Every gist created is a tiny repository. You can update gists and see the history using `git log`. You can download gists, hack on the repository, and `git push` them back into

the repository on gist.github.com (which will republish them onto the publicly facing web page). And, you can "fork" gists, just like any other repository.

You are allowed to branch within gist repositories; however, branches are not displayed inside of gist.github.com. But, if you need the benefits of branching when using GitHub gists you can branch normally inside a repository and keep the branch information on the upstream repository after you push it up.

You can have an unlimited number of public and secret gists. Instead of creating a new private repository from your limited amount in a paid GitHub account, you can take a tiny bit of code and make a secret gist, sharing this with others through a URL instead of the more onerous process of adding collaborators to a regular repository. Or, you can make a gist public, and share that URL to mailing lists or anywhere you need public feedback.

As there are two types of gists (public and secret), it is important the understand the differences between them. Public gists are searchable. Secret gists are not searchable, but they are accessible to anyone who knows the URL. Don't post any code to gist which you need to keep secret as once you put it there, it is only as safe as the URL is secret.

Most people share gists through the URL. But, you can embed gists inside of other contexts (like blogs) and get a simple and pretty snippet of code.

Embedding Gists Inside HTML

To embed inside of an HTML page look for the "Embed this gist" box to the left of a gist. Copy the code listed there (which will look something like `<script src="https://gist.github.com/xrd/8923697.js"></script>` and paste it into your HTML.

If you wish to include only a particular file from the Gist (if it contains multiple files), then add `?file=hi.rb` to the end of the URL specified in the src attribute.

Embedding Inside Jekyll blogs

Though we have not yet explained how Jekyll works (the GitHub blogging tool), it seems valid to point out the ease in which you can publish gists into a blog if that blog happens to be Jekyll hosted on GitHub.

Jekyll supports a fast shortcut code to embed a public gist inside of your Jekyll blog hosted on GitHub, or on any site built on the "github-pages" branch mechanism (described in the Chapter 6 chapter). The shortcut `{% gist 8138797 %}` will embed a private gist which would be found at `http://gist.github.com/8138797`.

If you want to use a specific file within the gist, add a filename to the gist code like `{% gist 8138797 hi.rb %}`.

Secret gists can also be embedded. If you use a secret gist, prefix the username of the account holder in the gist like so: `{% gist xrd/8138797 hi.rb %}`.

Gist from the command line

`gem install gist` will install a command line tool which assists in creating gists. You can use it simply by typing the command, and then entering the data you want to post as a gist.

```
$ gist
(type a gist. <ctrl-c> to cancel, <ctrl-d> when done)
{ "foo" : "bar" }
https://gist.github.com/9106765
```

The gist command will return the link to the gist just created. Gists are created anonymously by default. You can login using the `--login` switch. Once you do this, your gists will be linked to your account.

```
$ gist --login
Obtaining OAuth2 access_token from github.
GitHub username: xrd
GitHub password:
2-factor auth code: 787878

Success! https://github.com/settings/applications
```

You can pipe text to the gist command to use the contents of that file.

```
$ echo '{ "foo" : "bar" }' | gist
https://gist.github.com/9106799
```

You can also `cat` a file to gist.

```
$ cat MyJavaFile.java | gist
https://gist.github.com/9345609
```

Gists are often used to show interesting or troublesome code and there are times when you don't want to display the entirety of a file. In this case the command line `grep` tool can be a useful; `grep` searches for a specific piece of code and with the right switches can include several lines of context around that code inside a gist. This command looks for the function `myFunction` inside the `MyJavaFile.java` file and then prints the next 20 lines of context and stores it as a gist.

```
$ grep -A 20 myFunction MyJavaFile.java | gist
https://gist.github.com/9453069
```

Adding the `-o` switch automatically opens the gist inside your default web browser. You can also copy the gist URL to the clipboard using the `-c` switch. Or, you can copy the contents of your clipboard into a gist using the `-P` switch.

There are many other fun features of the gist command. To learn more run the gist command with the `--help` switch.

As gists are themselves repositories, you can use them for dual purposes: for hosting code samples, code samples which are themselves fully working and packaged applications inside a Git repository.

Gists as fully functioning apps

To demonstrate this ability, let's build a simple Sinatra application. Sinatra is a ruby library for creating dead-simple web servers. A Sinatra program looks as simple as this:

```
require 'sinatra'

get '/hi' do
  "Hello World!"
end
```

Create a gist for this by visiting gist.github.com. Enter in the text exactly as above and then choose public gist.

You now have a shareable gist of code, which anyone can use to review. More importantly, this is an executable piece of code. To use it, click into the "Clone this gist" box to the left of the body of content. You'll get a URL which looks something like this:

https://gist.github.com/8138797.git

Copy this and then enter a terminal program and enter this command:

```
$ git clone https://gist.github.com/8138797.git
$ cd 8138797
```

Now, you are inside the gist repository. If you look inside the repository you'll see a list of files, a list which right now numbers only one file.

```
$ ls
hi.rb
```

To run this code, enter `ruby hi.rb`

If you had not used Sinatra with ruby before, this will cause an error. This program requires a library called "sinatra" and you have not yet installed it. We could write a README, or add documentation into this file itself. Another way to guarantee the user has the proper files installed is to use a "Gemfile" which is a file that tells which libraries are installed and from where. That sounds like the best way:

```
$ printf "source 'https://rubygems.org'\ngem 'sinatra'" > Gemfile
```

The `bundle` command (from the bundler gem) will install Sinatra and the associated dependencies.

```
$ bundle
Using rack (1.5.2)
Using rack-protection (1.5.1)
Using tilt (1.4.1)
Using sinatra (1.4.4)
Using bundler (1.3.5)
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

Why did we do things this way? Because now we can add the Gemfile to our repository locally, and then publish into our gist for sharing on the web. Our repository now not only has the code, but a well know manifest file which explains the necessary components when running the code.

To publish our changes back into our gist from the command line, we need to update the "remote" repository reference. When we first cloned the repository we used the https link. If we are using SSH keys (and you should be; read the section "When should I use SSH vs HTTPS?") then we need to switch to use the SSH URL format (git protocol). Run this command:

```
$ git remote -v
origin      https://gist.github.com/8138797.git (fetch)
origin      https://gist.github.com/8138797.git (push)
```

Your results will be slightly different, but this output displays our remote repository; in other words, where we pull and push our code changes. If you are familiar with the way that remotes work on GitHub you can see that this is a read-only URL. We need to adjust these URLs in our remote so that it points to the read-write remote URL. To do that, remove the `https://` part and add a `git@`. Then, change the first `/` character after the `gist.github.com` URL to a `:` character. If your remote was the same as above you would have this `git@gist.github.com:8138797.git`. Then, in a terminal window run these commands:

```
$ git remote rm origin
$ git remote add origin git@gist.github.com:8138797.git
```

Now you can push new content in via the command line as well as edit files inside of GitHub.com. As you develop your app you have flexibility in whichever way fits you best.

Now let's take it further: what if we modified our application to use the GitHub API, specifically to access information about gists for a user?

Gists that render Gists

Let's add to our application and use the octokit gem to pull all public gists for any user we specify. Why would we want to make a gist that displays other gists? Self-referential meta code is all the rage, the modern day response to René Magritte's famous work: "Ceci n'est pas une pipe.".[1]

Add a view `index.erb` at the root of our directory.

```
<html>
<body>

User has <%= count %> public gists

</body>
</html>
```

Add the octokit gem to our Gemfile:

```
gem "octokit"
```

Run `bundle` to install octokit. Then, modify our hi.rb app to look like this:

```
require 'sinatra'
require 'octokit'

set :views, "."

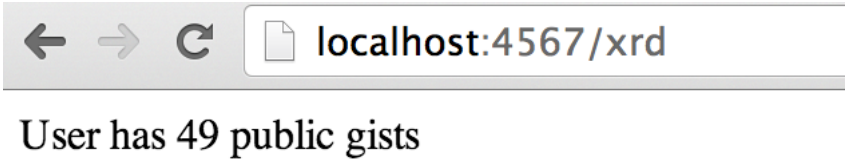
get '/:username' do |username|
  user = Octokit.user username
  count = user.public_gists
  erb :index, locals: { :count => count }
end
```

Our filesystem should look like this, with three files.

```
$ ls -1
Gemfile
hi.rb
index.erb
```

Run `bundle` to install octokit and restart Sinatra by running ctrl-c, and then `ruby hi.rb`. If you visit `http://localhost:4567/xrd` in your browser, you will see the count of public gists for user `xrd`; modify the username in the URL to any specify any GitHub username and you will see their last five gists displayed.

1. Explained best by Ben Zimmer *http://www.bostonglobe.com/ideas/2012/05/05/dude-this-headline-meta-dude-this-headline-meta/it75G5CSqi82NtoQHIucEP/story.html?camp=pm*



Going deeper into the Gist API

The GitHub API uses hypermedia instead of basic resource driven APIs. If you use a client like Octokit, the hypermedia details are hidden behind an elegant ruby client. But, there is a benefit to understanding how hypermedia works when you need to retrieve deeper information from the GitHub API.

Most RESTful APIs come with a "sitemap", generally a API reference document which tells a user which endpoints to use. You view the resources available from that API and then apply some HTTP verb to do something to them. Hypermedia thinks of an API differently. Hypermedia APIs describe themselves inside their responses using "affordances." What this means is that the API might respond like this:

```
{
    "_links": {
        "self": {
            "href": "http://shop.oreilly.com/product/0636920030300.do"
        }
    }
    "id": "xrd",
    "name": "Chris Dawson"
}
```

In this payload, you can see that there is an id ("xrd") and a name ("Chris Dawson"). Most APIs offer JSON responses, and this one does too. This particular payload was forked from the HAL explanation at the HAL Primer document and you can find a more detailed explanation of these concepts there.

The important thing to note about Hypermedia APIs is that payloads contain metadata about data itself and metadata about the possible options of operating on the data. RESTful APIs typically provide a mapping outside of the payload. You have to join the API sitemap with the data in an ad-hoc way when using RESTful APIs; with Hypermedia APIs your client can react to the payload itself correctly and intelligently without knowing anything about a sitemap stored in human readable documentation.

This loose coupling makes APIs and their clients flexible. In theory, a Hypermedia API works intuitively with a Hypermedia aware client. If you change the API, the client, as

it understands Hypermedia, can react and still work as expected. Using a RESTful API means that clients must be updated (either a newer version of the client must be installed) or the client code must be upgraded. Hypermedia APIs can alter their backend and the client, as long as it is hypermedia-aware, can automatically and dynamically determine the right way to access information from the response itself. In other words, with a hypermedia client the API backend can change and your client code should not need to.

This is explained in great detail in the book Building Hypermedia APIs with HTML5 and Node.

In the case of Octokit, navigating hypermedia looks like this:

- Start at a resource, with code like `user = Octokit.user "xrd"`. This begins the initialization of the client.
- `user` now is an object filled with the actual data of the resource. In this case, you could call a method like `user.followers` to see a meager follower count.
- `user` also has hypermedia references. You can see these by calling `user.rels`. This retrieves the relationships described in the hypermedia links. In this case, calling `.rels` shows a map of relationships, displayed in ruby code like: `#<Sawyer::Relation::Map: [:avatar, :self, :html, :followers, :following, :gists, :starred, :subscriptions, :organizations, :repos, :events, :received_events]>`
- Using one of these relationships starts by keying into the relationship hash and then using the get and data methods to request that information from the GitHub API: `followers = user.rels[:followers].get.data`.
- Once you call `.get.data` you will have a new followers object populated with an array of the followers (paged if it exceeds 100 items).

Let's extend our Sinatra app to retrieve actual data about the user's gists by using hypermedia references.

```
require 'sinatra'
require 'octokit'

set :views, "."

helpers do
  def h(text)
    Rack::Utils.escape_html(text)
  end
end

get '/:username' do |username|
  gists = Octokit.gists username, :per_page => 5
```

```
  erb :index, locals: { :gists => gists, username: username }
end
```

The `index.erb` file contains code to iterate over each gist and pull the content. You can see that our response object is an array of gists, each which has an attribute called `fields`. This fields attribute specifies the filenames available in each gist. If you reference that filename against the files, the response includes a hypermedia `ref` attribute. You can use this retreive the `raw` content using the Octokit method `.get.data`.

```
<html>
<body>

<h2>User <%= username %>'s last five gists</h2>

<% gists.each do |g| %>
<% g[:files].fields.each do |f| %>
<b><%= f %></b>:

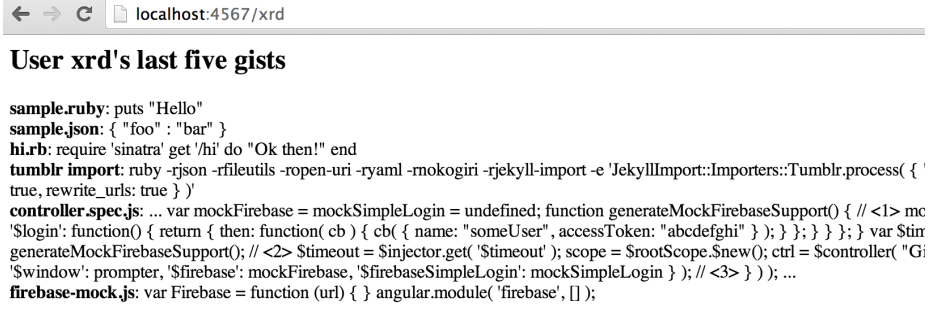
<%= h g[:files][f.to_sym].rels[:raw].get.data %>

<br/>
<br/>

<% end %>
<% end %>

</body>
</html>
```

Now we see the gists and the contents.



Summary

In this chapter we looked at gists and learned how they can be used to share code snippets. We built a simple application and stored it as a gist. This application retrieves data from the GitHub API using our first higher level language client library (the Octokit

library for Ruby). We also went deeper into how Hypermedia works and how a client library implements using Hypermedia metadata.

In the next chapter we will look at Gollum, the GitHub wiki. This chapter provides an introduction to the Rugged Ruby library for accessing Git repositories and the Ruby library for accessing GitHub.

CHAPTER 3

Gollum

Wikis have revolutionized the way we create and digest information. It turns out they are a great complement to technical projects (code repositories) because they allow non-technical users to contribute information without disturbing developers. Gollum is GitHub's open source version of a wiki. Just as Git has revolutionized collaborative editing of code, Gollum wikis layer the benefits of Git onto a proven publishing workflow. The true power of Gollum wikis reveal themselves when you see how tightly integrated with GitHub they are. You can quickly build and associate a wiki with any repository, and create a collaborative documentation system around any repository hosted on GitHub. And, you can pull in information from git repositories with ease, linking documentation with live code.

In this chapter we'll explore the basics of using Gollum, creating a wiki on GitHub and then understanding how to edit it on GitHub, and as a repostory on our local machine. We will then create a Gollum wiki by hand from the command line, and show the bare minimum set of files to call something a Gollum repository. Finally, we will build a simple image organization tool which allows us to edit a Gollum wiki in an entirely different way, but still publishes information into GitHub as a regular Gollum wiki, exploring a little bit of the internals of Git along the way.

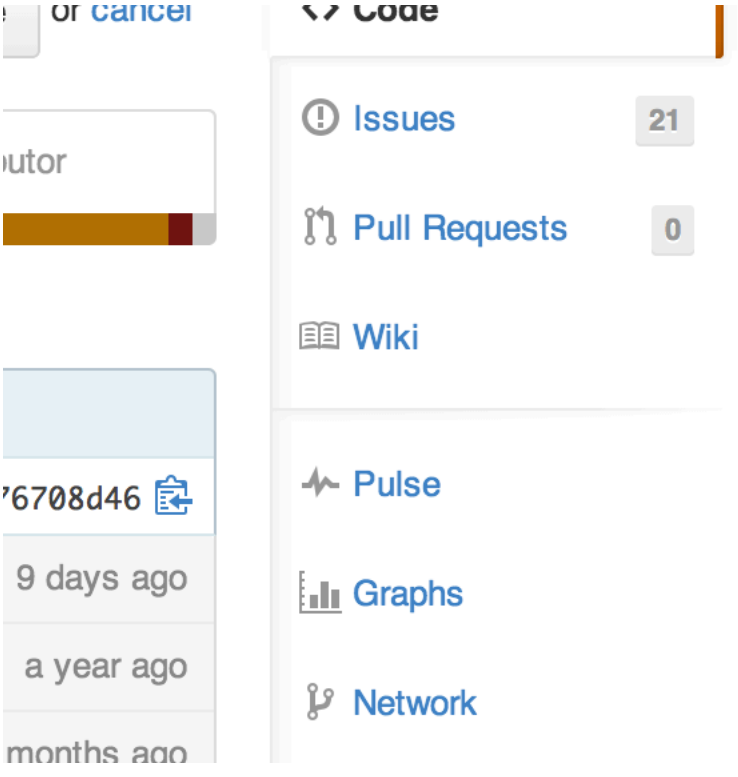
"The Story of Smeagol…"

Gollum wikis are simply an agreed upon file structure. At its most basic form, a Gollum wiki is a git repository with a single file, `Home.ext` (`ext` would be any of the supported wiki markup formats, which we will talk about later).

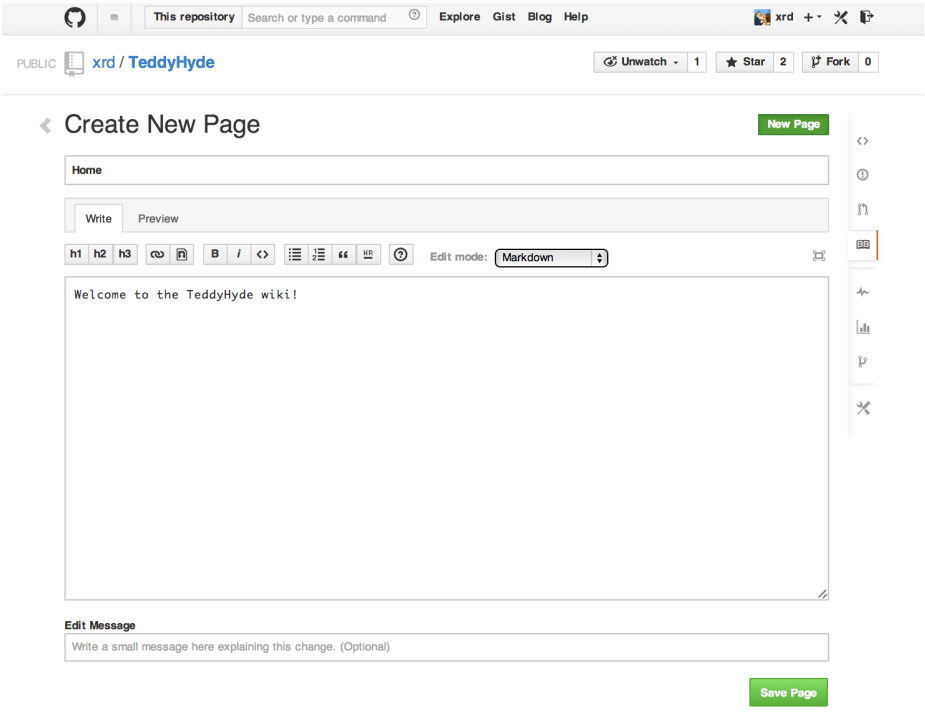
First, let's learn how to create a gollum wiki from the GitHub interface, and then later we'll move into creating one from scratch as a git repository.

Repository Linked Wikis

Any repository, public or private, can have an associated Gollum wiki. To create a wiki linked to a GitHub repository, visit the repository page and then look in the rightmost colum. You'll see an icon which looks like a book, next to which will be the word "Wiki."



Clicking on this link will bring you to a page where you are asked to create a wiki for the first time. GitHub will ask you to create the "Home" page, which is the starting point in a Gollum wiki. GitHub will automatically create a page template with the project name; you can customize this information to suit your own needs. Clicking on the "Save Page" will save your first page and create the wiki for you.



Your wiki is now as public as your repository is public. Public repositories have public wikis, accessible to anyone. Private repositories have private wikis, accessible only to those users or organizations which have rights to edit the repository data.

Wikis are powerful collaboration tools because they use a special language which compiles into HTML. Let's review the options for Gollum wikis.

Markup and Structure

Gollum files can be written in any of the supported "Github Markup" formats, which includes ASCIIdoc, Creole, Markdown, Org Mode, Pod, RDoc, ReStructuredText, Textile, and MediaWiki. The variety of markup languages brings flexibility but it can be confusing to know which one to use. Markdown (and its variant cousins) is the most popular markup language on GitHub, and is well liked on other popular sites like Stack Overflow. If you are unsure which language to use, Markdown is a safe bet because it is ubiquitous across GitHub. The Chapter 6 chapter has a much deeper overview of Markdown.

If you do choose Markdown, In addition to the standard vanilla Markdown language tags, Gollum adds its own set wiki specific tags. There are often subtle (or conflicting)

differences from other Wiki markup so it is worth reviewing the Gollum repository documentation page. We'll go over the most important ones here.

Links

Let's summarize a bit of Gollum syntax. Links are a good place to start.

- Links use the syntax `[[Link]]`.
- You can add a link title using the bar character: `[[http://foobar.com|A link to foobar]]`.
- Links can be either external or internal links.
- A link like `[[Review Images]]` will be converted to a relative link to the page `review-images.ext` (where ext is the preferred extension you are using with your wiki, most likely Markdown).

Wikis are generally a collection of pages linked together in myriad ways, and this assumption about the structure of links makes writing pages easier.

As we mentioned, there are differences between Gollum wiki tags and other wikis despite having similar syntax. One such example is MediaWiki, where links with titles use the opposite ordering `[[A link to foobar|http://foobar.com]]`, so caveat emptor.

Inserting Images

Images are inserting into your document using the same tag format `[[ceo.png]]`. Overloading this syntax makes it simple to remember the basics of adding relationships inside your wiki, whether they be images or page links. There are many options you can provide when generating image tags. For example, to add a frame and an alt tag, you could use syntax like `[[ceo.png|frame|alt=Our CEO relaxing on the beach]]`. Review the documentation on the Gollum repository for more details about the breadth of the image options.

Code Snippets

Gollum (the wiki) was invented at GitHub, a company dedicated to improving the lives of software developers, so it stands to reason Gollum wikis would support code snippets. To include a snippet of code, use three backticks, followed by the language name, and close the block of code using three more backticks.

```
```ruby
def hello
  puts "hello"
end
```
```

A more interesting way of embedding code inside a Gollum repository is to use the file include syntax. Again, use a triple backtick, followed by the file type and then a reference to the code snippet inside a GitHub repository. You'll need to include the branch as well.

```
```html:github:xrd/TeddyHyde/blob/master/Gemfile```
```

This will pull the file on GitHub located inside the "TeddyHyde" repository for user "xrd" on the master branch named "Gemfile" and publish it with syntax hilighted into your wiki as if you had used this markup.

```
```ruby
source 'https://rubygems.org'

gem "nokogiri"
gem "rspec"
gem 'calabash-android', :git => 'git://github.com/calabash/calabash-android.git'
```
```

Unfortunately, you cannot specify a specific SHA commit to retrieve moments in history within the git repository, but this is still a powerful way to expose a file inside a Gollum wiki. If you need to do that, the best way might be to create a branch from a specific SHA commit, and then reference that branch when including the file.

```
$ git checkout 0be3e4475db2697b8
$ git checkout -b at_sha_0be3e4475db2697b8
$ echo "gem 'rails' # Adding rails to Gemfile" >> Gemfile
$ git commit -m "Added rails to Gemfile" -a
$ git push origin at_sha_0be3e4475db2697b8
```

This would generate a new branch based on an older commit, and push up the branch. Then, you could reference this inside your wiki with the following include

```
```html:github:xrd/TeddyHyde/blob/at_sha_0be3e4475db2697b8/Gemfile```
```

Note that we've referenced the branch named after the specific SHA hash we want.

Gollum Structural Components

Gollum includes capabilities to add sidebars, headers, and footers. If you include a file called `_Sidebar.ext` inside your repository, you'll see it as a sidebar for every file rendered. Sidebars are automatically added to any file and any file from subdirectories that do not have their own sidebar files. If you wanted to add sidebars specific to a subdirectory, add another file in the subdirectory and this file will override the top level sidebar file.

No Styling or JavaScript

Finally, for security reasons, Gollum strips out all CSS and JavaScript from raw markup files. You can include your own JavaScript or CSS file when running Gollum locally

using the `--custom-css` or `--custom-js` switches, but there is no way to include these files on a Wiki when your Gollum wiki is hosted on GitHub.

Now that we have investigated the structure and format of using Gollum wikis, we can dig into the power tools that come with Gollum.

Moving Gollum to Your Laptop

Though many people edit Gollum wikis exclusively from within the GitHub online editor, there is a real flexibility and power when hosting your wiki locally and editing it from your laptop. To do this you need to install the command line tools for Gollum.

```
$ gem install gollum
```

You will then see the `gollum` command in your path.

There is a difference between the `gollum` command (what you run from the command line) and the suite of technologies that make up Gollum as a project. To differentiate between them, remember that we are talking about the suite when the word is capitalized ("Gollum"), and the command line tool when the word is lowercased and fixed width font (`gollum`).

What additional options are opened up when running locally? Let's take a deeper look.

Alternative Editing Options

When you run your Gollum wiki from your laptop instead of viewing and editing on GitHub exclusively, you then get a few additional options for editing.

- Gollum wikis are editable locally within a web browser: run the `gollum` command locally on your laptop (which spawns a mini web server hosting your wiki) and then browse to *http://localhost:4567*.
- Gollum wikis can be edited at the file system level using your favorite editor, allowing you the flexibility of staying within the same workflow you use to edit any other file within a local repository.

You might use a hybrid approach to editing your Gollum wiki, switching back and forth between editing within the web browser interface and jumping into the command line and using your editor to create or modify files when you need to use "power commands." And, it is nice to know you can use any simple text processing language to make programmatic changes to your wiki once you have it locally on your laptop as a Git repository.

Editing with the Command Line

Gollum wiki content reflects only the files inside the repository; another way to say this is that files in your working directory but not yet committed are not used by Gollum. To illustrate this, let's go through the steps to add a sidebar to our wiki. Adding a sidebar means you need to create a file called _Sidebar.md. This is a special file which Gollum recognizes and generates a sidebar for you; the first part of the name is fixed, but you can change the extension to whatever extension you prefer for your wiki files. If we use the "open" command (available on Mac or Linux) to open the Wiki in our default browser, you will see that only once we have committed the change to our repository do we actually see the sidebar.

```
$ gollum & # Start gollum, will run on port 4567.
$ printf "## Warning\nPlease note this is subject to change" > _Sidebar.md
$ open http://localhost:4567/ # We won't see our sidebar yet...
$ git add _Sidebar.md
$ open http://localhost:4567/ # We still don't see our sidebar...
$ git commit -m "Add sidebar"
$ open http://localhost:4567/ # Eureka, now we will see our sidebar!
```

If you edit within the web browser interface, the underlying Grit libraries do all the work to commit new files into the repository. If you use the command line, you'll need to remember to commit files manually to see them inside your wiki.

We now can display and allow editing from our locally running Gollum server. This Gollum server actually can be made to be accessible to anyone who can access your laptop's IP address. So, we could permit editing by others in our office. But, what if we want to disallow editing unless the user has permission to do so? We need an authentication mechanism.

Adding Authentication

As Gollum is built on top of Sinatra (a ruby library which we will discuss in more detail shortly) you can utilize authentication gems like `omnigollum` or write your own authentication handler and run gollum inside that. This thread on StackOverflow explains how to install your own handler:

```
# authentication.rb
module Precious
  class App < Sinatra::Base
    use Rack::Auth::Basic, "Restricted Area" do |username, password|
      [username, password] == ['admin', 'admin']
    end
  end
end
```

And, then run it using this command:

```
gollum --config authentication.rb
```

You'll then be prompted for the username and password, and can use "admin" and "admin".

Why "Precious" as the module name? If you peek into the Gollum code itself, you'll see that the Gollum libraries use a namespace called "Precious" (the word used to refer to the ring by the character Gollum in the Lord of the Rings books) as the base class for the Gollum wrapper around Sinatra. This code extends the instance of Sinatra running your Gollum wiki and adds an authentication layer.

A word of caution when using the gollum command in server mode to edit files locally inside a web browser. If you start the gollum server from the command line you do have the capability to edit the files from any computer within the same network. In other words, you could find your IP address and use that address from your Chromebook or your tablet to view and edit your wiki. However, remember that the gollum server command does not have an authentication system built into it, which means that gollum thinks anyone accessing the wiki is the same user that started the gollum command. This is fine if you are in the other room editing on your tablet while gollum runs on your work laptop. However, the gollum server is not a good solution for offering a wiki up to many people within a subnet. If multiple people edit files, there is no way that gollum can track the different user contributions in the change log. This is not a problem when editing your Gollum wiki inside GitHub.com: the GitHub site knows who you are and properly assigns your changes to your username inside the change log.

We've played a bit with the gollum command line tools. Let's put these skills to use and make our own special gollum tool.

Building a Gollum Editor

Once you understand Git repositories, you can see the power of Gollum as a wiki format: as everything is built on Git, you can manage your wiki using all the great tools that come with Git. We've explored how easy it is to edit Gollum wikis: from within the command line, from the web browser, or from within GitHub. However, there might be times when you need to provide an entirely customized editing modality. As long as you write files into the repository in the way the gollum libraries understand, you can write your own editing interface to suit your own needs. Let's experiment with this idea and build a new editing interface for Gollum wikis.

Gollum is a tool that provides flexibility by allowing local usage: this can be very handy when you are on a plane and don't want to pay for Wi-Fi. However, at the time of this writing there is a bug where images are not displayed, so although you can fully edit files using the local `gollum` command, you will not be be able to view them when viewing your wiki on you local machine. To view image files correctly, publish them into GitHub.

Hacking Gollum

Team software development often revolves around this idealized scenario: a business person develops the structure of the application with higher-up stakeholders, these ideas are passed down to a UI/UX designer who then creates wireframes and mockups of the interactions, and then a software developer takes these wireframes and builds the software. Put another way, program managers figure out what features provide the most value to users, which then trickles down into the wireframes as real interactions. Many hidden blocking paths are fleshed out here, places where the application would confuse the user, and time is saved because the software developer does not have to waste time building something that would not work anyway. By the time it reaches the developer, the UI interaction is streamlined and the costly and inflexible stage of building software has all the inefficiencies optimized away. The developer can simply work on a piece of software and know there are no changes, changes which would be extremely costly to implement.

In practice, this process is almost never so smooth. What typically happens is the business people don't completely understand all the requirements when they document the structure they want, so after they have committed to a structure they later ask for changes, which trickle down into the designs. The "final and approved" mockups have to be changed and this then needs to be communicated to the developer, who has already started building something that was "set in stone." Or, the developer, as she is building the software, realizes there are missing steps to get to places inside the application, and needs to communicate this back to the designer. If you have multiple people doing software development on a project, this information then needs to fan out to them if their areas are affected by these changes. This information must traverse many different people, with many different methods of communication.

Wikis are a great way to store this kind of information. Information which changes. Information which must be retrieved by many people and edited by many people. What better than to manage these informational transitions than a change tracking system like Git, and what better way to absorb this information than a Wiki built on top of Git, hosted on GitHub.

Wireframe Review Tool

Let's build a simple tool which stores these types of changes. We'll build an image editor that hosts changes to UI mockups. This will give our executives a place where they can see changes and updates. This will allow our UI designer a place to store their images and annotate them with vital information. And, we'll have a place where developers can retrieve information without reviewing their email and wondering "Do I have the most up-to-date mockups?" We'll buid a special interface which allows quickly editing and reviewing these files locally. And all of it can be published into GitHub for review (though we won't allow editing of the information there, since GitHub has its own editing modality.)

Gollum is built on Ruby and uses the Grit library underneath. Using Ruby makes sense because we can leverage the existing Grit and Gollum libraries. We'll also use Sinatra, the same web application library we used in the last chapter.

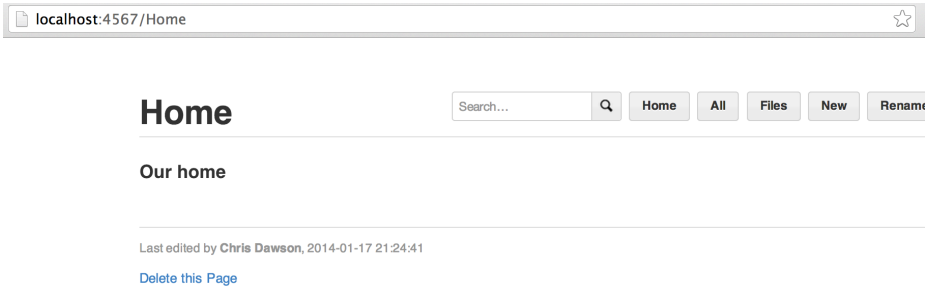
The `gollum` command is, in fact, a customized wrapper around Sinatra.

This will be a dual purpose repository. We can use the repository with gollum as a standard wiki. And, we can use it with our application to enter data in a more powerful way than gollum permits from its default interface. The data will still be compatible with gollum and will be hosted on GitHub.

To begin, initialize our repository.

```
$ mkdir images
$ cd images
$ git init .
$ printf "### Our home" > Home.md
$ git add Home.md
$ git commit -m "Initial checking"
```

We've just created a wiki compatible with gollum. Let's see what it looks like inside gollum. run the `gollum` command then open `http://localhost:4567/` in your browser.



As you can see, this tiny set of commands was enough to create the basics of the gollum wiki structure.

Create our sinatra script called `image.rb`, and then we can install the necessary gems and run our server application.

```
require 'sinatra'
require 'gollum-lib'
wiki = Gollum::Wiki.new(".")
get '/pages' do
  "All pages: \n" + wiki.pages.collect { |p| p.path }.join( "\n" )
end
```

```
$ printf "source 'https://rubygems.org'\n\ngem 'sinatra'\ngem 'gollum-lib'" >> Gemfile
$ bundle install
$ ruby image.rb
$ open http://localhost:4567 # or whatever URL is reported from Sinatra
```

Once you open this in your browser, you'll see a report of the files that exist in our Gollum wiki right now. We've only added one file, the `Home.md` file.

Programmatically Handling Images

Let's add to our server. We want to support uploading ZIP files into our system that we will then unpack and add to our repository, as well as adding a list of these files to our wiki. Modify our image.rb script to look like this:

```
require 'sinatra'
require 'gollum-lib'
require 'tempfile'
require 'zip/zip'

def index( message=nil )
  response = File.read(File.join('.', 'index.html'))
  response.gsub!( "<!-- message -->\n", "<h2>Received and unpacked #{message}</h2>" ) if message
  response
```

```
end

wiki = Gollum::Wiki.new(".")
get '/' do
  index()
end

post '/unpack' do
  @repo = Rugged::Repository.new('.')
  @index = Rugged::Index.new

  zip = params[:zip][:tempfile]
  Zip::ZipFile.open( zip ) { |zipfile|
    zipfile.each do |f|
      contents = zipfile.read( f.name )
      filename = f.name.split( File::SEPARATOR ).pop
      if contents and filename and filename =~ /(png|jp?g|gif)$/i
        puts "Writing out: #{filename}"
      end
    end
  }
  index( params[:zip][:filename] )
end
```

We'll need an `index.html` file as well, so add that.

```
<html>
<body>
<!-- message -->
<form method='POST' enctype='multipart/form-data' action='/unpack'>
Choose a zip file:
<input type='file' name='zip'/>
<input type='submit' name='submit'>
</form>
</body>
</html>
```

This server script receives a POST request at the `/unpack` mount point and retrieves a ZIP file from the parameters passed into the script. It then opens the ZIP file (stored as a temp file on the server side), iterates over each file in the ZIP, strips the full path from the filename, and then prints out that filename (if it looks like an image) to our console. Regardless of whether we are accessing the root of our server, or have just posted to the `/unpack` mount point, we always need to render our index page. When we do render it after unzipping, we replace a comment stored in the index file with a status message indicating the script received the correct file we posted.

We need to add an additional ruby library to enable this application, so update the required gems using the following commands, and then re-run our Sinatra server script.

```
$ printf "gem 'rubyzip'\n" >> Gemfile
$ bundle install
$ ruby image.rb
```

Then, we can open `http://localhost:4567/` and test uploading a file full of images. You'll see output similar to this in your console after uploading a file.

```
...
[2014-05-07 10:08:49] INFO  WEBrick 1.3.1
[2014-05-07 10:08:49] INFO  ruby 2.0.0 (2013-05-14)
[x86_64-darwin13.0.0]
== Sinatra/1.4.5 has taken the stage on 4567 for development with
backup from WEBrick
[2014-05-07 10:08:49] INFO  WEBrick::HTTPServer#start: pid=46370
port=4567
Writing out: IMG1234.png
Writing out: IMG5678.png
Writing out: IMG5678.png
...
```

Leveraging the Rugged Library

Our end goal for this script is to add files to our Gollum wiki, which means adding files to the repository which backs our Gollum wiki. The Rugged library handles the grunt work of this type of task easily. Rugged is the successor to the original Ruby library for Git (called Grit). Gollum, at the time of this writing uses the Grit libraries, which also provide a binding to the libgit2 library, a "portable, pure C implementation of the Git core methods." Grit has been abandoned (though there are unofficial maintainers) and the Gollum team intends to use Rugged as the long term library backing Gollum. Rugged is written in Ruby and, if you like Ruby, is a more elegant way to interface with a Git repository than raw git commands. As you might expect, Rugged is maintained by several employees of GitHub.

To change our script to modify our Git repository, modify the `puts` statement inside the zip loop to call a new method called `write_file_to_repo`. And, at the end of the zip block, add a method called `build_commit` which builds the commit from our new files. Our new file (omitting the unchanged code at the head of the file) looks like this.

```
post '/unpack' do
  @repo = Rugged::Repository.new('.')
  @index = Rugged::Index.new

  zip = params[:zip][:tempfile]
  Zip::ZipFile.open( zip ) { |zipfile|
    zipfile.each do |f|
      contents = zipfile.read( f.name )
      filename = f.name.split( File::SEPARATOR ).pop
      if contents and filename and filename =~ /(png|jp?g|gif)$/i
        write_file_to_repo contents, filename # Write the file
      end
    end
    build_commit() # Build a commit from the new files
  }
```

```
  index( params[:zip][:filename] )
end

def get_credentials
  contents = File.read File.join( ENV['HOME'], ".gitconfig" )
  @email = $1 if contents =~ /email = (.+)$/
  @name = $1 if contents =~ /name = (.+)$/
end

def build_commit
  get_credentials()
  options = {}
  options[:tree] = @index.write_tree(@repo)
  options[:author] = { :email => @email, :name => @name, :time => Time.now }
  options[:committer] = { :email => @email, :name => @name, :time => Time.now }
  options[:message] ||= "Adding new images"
  options[:parents] = @repo.empty? ? [] : [ @repo.head.target ].compact
  options[:update_ref] = 'HEAD'

  Rugged::Commit.create(@repo, options)

end

def write_file_to_repo( contents, filename )
  oid = @repo.write( contents, :blob )
  @index.add(:path => filename, :oid => oid, :mode => 0100644)
end
```

As you can see from the code above, Rugged handles a lot of the grunt work required when creating a commit inside a Git repository. Rugged has a simple interface to creating a blob inside your Git repository (`write`), adding files to the index (the `add` method), and then has a simple and clean interface to build the tree object (`write_tree`) and then build the commit (`Rugged::Commit.create`).

The astute observers among you will notice a method called `get_credentials` which loads up your credentials from a file located in your home directory called `.gitconfig`. You probably have this if you have used Git for anything at all on your machine, but if this file is missing, this method will fail. On my machine this file looks like the following code snippet. The `get_credentials` method simply loads up this file and parses it for the name and email address. If you wanted to load the credentials using another method, or even hard code them, you can just modify this method to suit your needs. The instance variables `@email` and `@name` are then used in the `build_commit()` method.

```
[user]
        name = Chris Dawson
        email = xrdawson@gmail.com
[credential]
        helper = cache --timeout=3600
...
```

Just to double check that everything worked properly, let's verify that things are working correctly after uploading a ZIP file. Jumping into a terminal window after uploading a new file, imagine running these commands:

```
$ git status
```

To our surprise, we will see something like this:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  deleted:    images/3190a7759f7f6688b5e08526301e14d115292a6e/IMG_20120825_164703.jpg
  deleted:    images/3190a7759f7f6688b5e08526301e14d115292a6e/IMG_20130704_151522.jpg
  deleted:    images/3190a7759f7f6688b5e08526301e14d115292a6e/IMG_20130704_174217.jpg
```

We just added those files; why is Git reporting them as deleted?

To understand why this happens, remember that in Git there are three places where files can reside: the working directory, the staging area or index, and the repository itself. Your working directory is the set of local files which you are working on. The `git status` command describes itself as "show the working tree status." Rugged operates on the repository itself, and our Rugged calls above operated on the index and then built a commit. This is important to note because our files will not exist in our working directory if we only write them using the Rugged calls, and if we do this, we cannot reference them inside our wiki page when we are running Gollum locally. We'll fix this in the next section.

We've now added the files to our repository, but we have not exposed these files inside our wiki. Let's modify our server script to write out each file to a wiki page for review. As we mentioned in the previous section, we need to make sure that we write the files to both the working index and the repository (using the Rugged library `write` call). Then we can generate a Review file which details all the images uploaded.

Optimizing for Image Storage

Often while a designer is receiving feedback from stakeholders, they will incorporate that feedback into the UI comps, and then resend the set of comps, with only a tiny change in one file, where the remaining dozens or even hundreds of files have been left unchanged. We might assume that our code, as it writes these files to a different path on disk inside the repository (the path is the parent SHA hash to make it unique) and we could therefore be adding the same file multiple times, and creating a big wasteful repository. However, the nature of git permits us to add the same file multiple times without incurring any additional storage cost beyond the first addition. When a file is

added to git repository, a SHA hash is generated from the file contents. For example, generating the SHA hash from an empty file will always return the same SHA hash.[1]

```
$ echo -en "blob 0\0" | shasum
e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
$ printf '' | git hash-object -w --stdin
e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

Adding a zip file with a bunch of files where only one or two differs from the prior zip file means that Git will properly reference the same file multiple times. Unfortunately, GitHub does not provide an interface for reviewing statistics of Wikis in the same way that they do for regular repositories. We can, however, review our repository size from within the local repository by running the count-objects Git subcommand. As an example, I uploaded a ZIP file with two images inside of it. I then use the count-objects command and see this:

```
$ git gc
...
$ git count-objects -v
count: 0
size: 0
in-pack: 11
packs: 1
size-pack: 2029
prune-packable: 0
garbage: 0
size-garbage: 0
```

Inspecting the first ZIP file, I see these statistics about it.

```
$ unzip -l ~/Downloads/Photos\ \(4\).zip
Archive:  /Users/xrdawson/Downloads/Photos (4).zip
  Length     Date   Time    Name
 --------    ----   ----    ----
  1189130  01-01-12 00:00   IMG_20130704_151522.jpg
   889061  01-01-12 00:00   IMG_20130704_174217.jpg
 --------                   -------
  2078191                   2 files
```

I then use another ZIP file which has one additional file, with the other two included files exactly identical.

```
unzip -l ~/Downloads/Photos\ \(5\).zip
Archive:  /Users/xrdawson/Downloads/Photos (5).zip
  Length     Date   Time    Name
 --------    ----   ----    ----
  1189130  01-01-12 00:00   IMG_20130704_151522.jpg
   566713  01-01-12 00:00   IMG_20120825_164703.jpg
   889061  01-01-12 00:00   IMG_20130704_174217.jpg
```

1. This is explained beautifully in the blog *http://alblue.bandlem.com/2011/08/git-tip-of-week-objects.html*.

```
--------                    -------
 2644904                    3 files
```

Then, I upload the second ZIP file. If I re-run the count-object command (after running `git gc`, a command which packs files efficiently and makes our output more human readable), I see this:

```
$ git gc
...
$ git count-objects -v
count: 0
size: 0
in-pack: 17
packs: 1
size-pack: 2578
prune-packable: 0
garbage: 0
size-garbage: 0
```

Notice that our packed size has only changed by about half a MB, which is the compressed size of the additional third file, but more importantly, there was no impact from the other two files on our repository size, even though they were added at different paths.

If we upload the secondary file yet again, we will regenerate and commit a new version of the `Review.md` file, but no new files will need to be created inside our Git repository object store from the images directory (even though their paths have changed), so our impact on the repository will be minimal.

```
$ git gc
...
$ git count-objects -v
count: 0
size: 0
in-pack: 21
packs: 1
size-pack: 2578
prune-packable: 0
garbage: 0
size-garbage: 0
```

As you can see, our packed-size has barely changed, an indication that the only changes were a new Git tree object and commit object. We still do have the files located in our repository at a variety of paths so our review pages will work no matter what revision we are accessing.

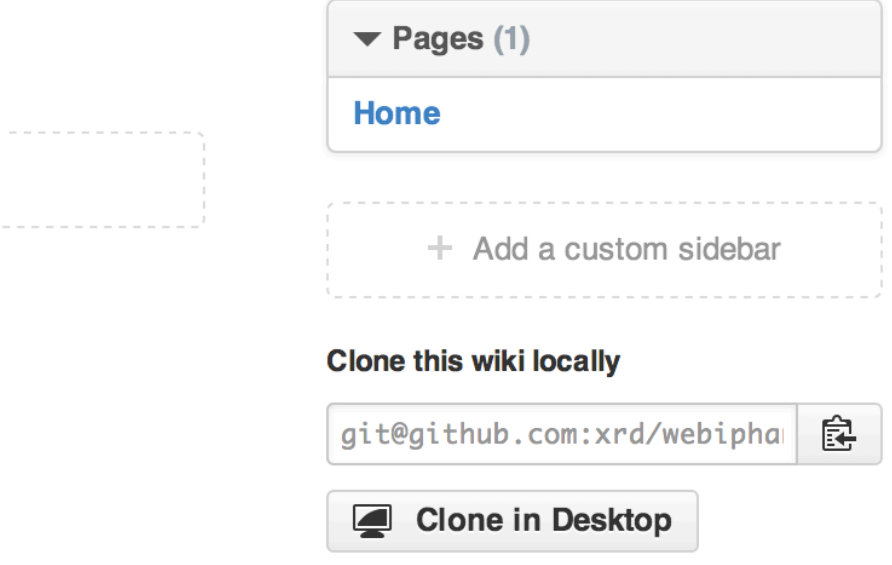
```
$ find images
images
images/7507409915d00ad33d03c78af0a4004797eec4b4
images/7507409915d00ad33d03c78af0a4004797eec4b4/IMG_20120825_164703.jpg
images/7507409915d00ad33d03c78af0a4004797eec4b4/IMG_20130704_151522.jpg
images/7507409915d00ad33d03c78af0a4004797eec4b4/IMG_20130704_174217.jpg
```

```
images/7f9505a4bafe8c8f654e22ea3fd4dab8b4075f75
images/7f9505a4bafe8c8f654e22ea3fd4dab8b4075f75/IMG_20120825_164703.jpg
images/7f9505a4bafe8c8f654e22ea3fd4dab8b4075f75/IMG_20130704_151522.jpg
images/7f9505a4bafe8c8f654e22ea3fd4dab8b4075f75/IMG_20130704_174217.jpg
images/b4be28e5b24bfa46c4942d756a3a07efd24bc234
images/b4be28e5b24bfa46c4942d756a3a07efd24bc234/IMG_20130704_151522.jpg
images/b4be28e5b24bfa46c4942d756a3a07efd24bc234/IMG_20130704_174217.jpg
```

Git and Gollum can efficiently store the same file at different paths without overloading the repository.

Reviewing on GitHub

The raison d'etre for this wiki is to annotate a development project. If you follow the instructions above and create a new wiki for a repository, you'll then be able to push up the changes we've made using our `image.rb` script. Once you have created a new wiki, look for a box on the right which says "Clone this wiki locally".

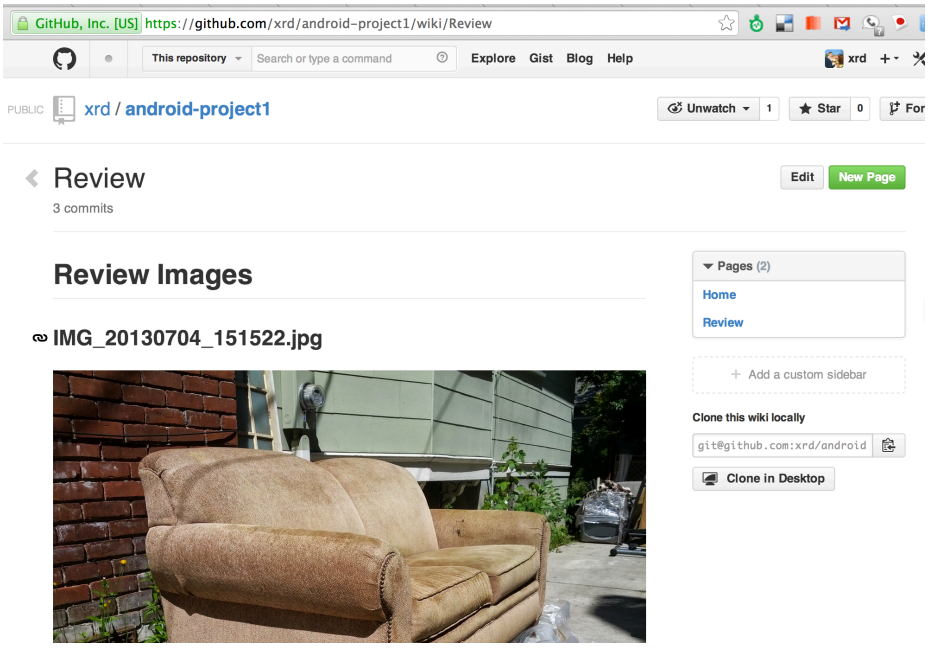


Copy that link, and then enter a terminal window where we can then add a remote URL to our local repository which allows us to synchronize our repositories and publish our images into GitHub. Gollum wikis have a simple URL structure based on the original clone URL: just add the word `.wiki` to the end of the clone URL (but before the fi-

nal `.git` extension). So, if our original clone URL of the repository is `git@github.com:xrd/webiphany.com.git` our clone URL for the associated wiki will be `git@github.com:xrd/webiphany.com.wiki.git`. Once we have the URL, we can add it as a remote to our local repository using the following commands.

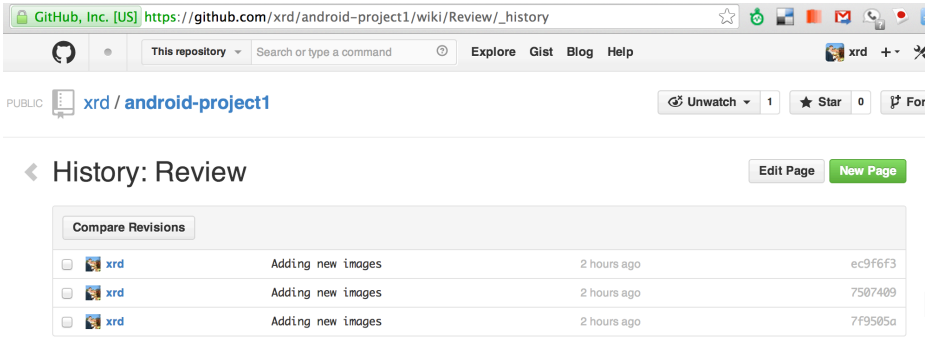
```
$ git remote add origin git@github.com:xrd/webiphany.com.wiki.git
$ git pull # This will require us to merge the changes...
$ git push
```

When we pull, we will be asked to merge our changes since GitHub created a `Home.md` file which did not exist in our local repository. We can just accept the merge as-is. The `git push` publishes our changes. If we then visit the wiki, we'll see an additional file listed under the pages sidebar to the right. Clicking on the review page, we can see the images we've added most recently.



Not sure why our designer is providing us with an image of a couch, but I am sure he has his reasons.

Once have published the file, we can click on the "Review" link in the sidebar to see the most current version of the "Review" page. We also can review the revisions of this file by clicking on the "3 Commits" (or whatever number of commits have occurred with this file). link right underneath the page title. Jumping onto that page shows us the full history of this file.



Clicking on any of the SHA hashes will display the page at that revision in our history and show us the state of the document at any given moment in history. Unfortunately, jumping back and forth between revisions requires two clicks, one from the review page to the list of revisions, and then another click to jump into the revision we want, but this permits us to review changes between the comps provided from our designer.

It would be nice if GitHub provided a simple way to jump from a revision to the parent (older) revision, but they don't expose this in their site as of this writing. We can fix this, however, by generating our own special link inside the review page itself which will magically know about how to navigate to a previous version of the page.

Improving Revision Navigation

In our example, we only have three revisions right now, and all share the same commit message ("Adding new images"). This is not very descriptive and makes it challenging to understand the differences between revisions, critical when we are trying to understand how things have changed between comps. We can improve this easily.

First, let's add a commit message field to our upload form.

```
<html>
<body>
<!-- message -->
<form method='POST' enctype='multipart/form-data' action='/unpack'>
Choose a zip file:
<input type='file' name='zip'/>
<input type='text' name='message' placeholder='Enter commit message'/>
<input type='submit' name='submit'>
</form>
</body>
</html>
```

Then, let's adjust the commit message inside our `image.rb` script, which is a one line change to the options hash, setting the value of it to the parameter we are now passing in for "commit".

```
  ...
options[:committer] = { :email => @email, :name => @name, :time => Time.now }
options[:message] = params[:message]
options[:parents] = @repo.empty? ? [] : [ @repo.head.target ].compact
  ...
```

Now, if our designer posts a new version of the UI comps, they can specify what changes were made, and we have a record of that in our change log, exposed on the revisions section of our wiki hosted on GitHub.

Fixing Linking Between Comp Pages

We noted that there is no quick way to jump between comps once we are inside a review revision. However, if you recall we used the parent SHA hash to build out our image links. We can use this to build out a navigation inside our comp page when we are on a revision page while viewing the history.

Again, it is a simple change: one line within the `write_review_file` method. After the block which creates each link to the image files, add a line which builds a link to the parent document via its SHA hash using the parent SHA found in our Rugged object under `@repo.head.target`. This link will allow us to navigate to prior revisions in our history.

```
  ...
files.each do |f|
  contents += "### #{f} \n[[#{dir}/#{f}]]\n\n"
end
contents += "[Prior revision (only when viewing history)](#{@repo.head.target})\n\n"

File.write review_filename, contents
oid = @repo.write( contents, :blob )
  ...
```

Now, when we view the Review file history, we see a file with a link at the bottom to the link to each prior version. Is it possible to provide a link to the next version in our history? Unfortunately, we have no way to predict the SHA hash of the next commit made to the repository, so we cannot build this link inside our `Review.md` file with our ruby script. However, we do get something just as good for free because we can simply use the back button to jump back to the prior page in the history stack of our browser. It would be nice if we could generate this link alongside the link we placed into the wiki markup, and we could do this using a link that runs an onclick handler delegating to a JavaScript command like `window.history.back()`, but Gollum foils us again by stripping JavaScript from our markup files as we noted before. This is a good thing generally, as we don't want to permit rogue markup inside our wiki pages, but it does limit our options in this situation.

Unfortunately, these links do not work when you are viewing the review file itself (clicking on them brings you to a page which asks you to create this as a new page). Gollum,

unlike Jekyll, does not support Liquid tags which would permit building a link using the username and repository. Right now we don't have access to these variables, so our link needs to be relative, which works when we are in history review, but not in the normal review. It does not affect viewing the files so this would require educating your stakeholders on the limitations of this link.

Summary

In this chapter we learned how to create a Gollum wiki from scratch, both on GitHub and as a fresh repository from the comman line. We then looked at the different ways to use the `gollum` command line tool and learned why this is a nice option when we want to run our own Gollum server. Finally, we built a customized Gollum image-centric editor using the Rugged library for Ruby.

Our next chapter explores the GitHub API from what might be an unexpected vantage point: JavaScript. In this chapter we will show you how to host an entire application on GitHub, no server required, that still allows modification of data inside of GitHub by its users.

CHAPTER 4

Python and the Search API

Once you have enough data, no amount of organization will make everything easy to find. As Google has taught us, the only system that works at this scale is a search box. When you use GitHub, you're exposed to both sides of this phenomenon: the repositories you have direct access to — which are relatively small in number — are given a single level of hierarchy, so you can keep them straight in your head. For the rest, the uncountable millions of public repositories that belong to other people, there's a search box, with powerful features to help you find what you're looking for.

Helpfully, GitHub also exposes this capability as an API you can consume from your own applications. GitHub's search API gives you access to the full power of the built-in search function. This includes the use of logical and scoping operators, like `or` and `user`. By integrating this feature with your application, you can provide your users a very powerful way of finding what they're looking for.

In this chapter we'll take a close look at this API, and try building something with it. We'll see how the search API is structured, what kind of results come back, and how it can help us create a feature for someone on our team.

General Principles

The search API is split into four separate parts:

- Repositories
- Code
- Issues
- Users

These APIs all have separate subject matter, and have different formats for their results, but they all behave the same in a few key ways. We're covering these first, because they'll

help you understand the results coming back from the specific API calls that we cover further down. There are four major areas of commonality.

Authentication

Your identity as a user can determine the result set from a search query, so it's important to know about authentication. We cover GitHub authentication fully in "Authentication" on page 11, but this API is also available without logging in. However, there are a few limitations to this approach.

First, you'll only be able to search public repositories. This is probably fine if you're primarily working with open-source software, but users of your application will probably expect to have access to their private code, as well as that of any organizations they belong to. Also, since *all* Enterprise repositories are private, anonymous search is completely useless there.

Secondly, authenticating opens up your rate limit. The limits on search are stricter than other APIs anyways, because search is computationally expensive, but anonymous searches are stricter still. As of this writing, and according to the documentation, anonymous searches are limited to 5 per minute, and you can do 20 authenticated queries per minute. Take a look at "GitHub API Rate Limits" on page 19 for more on how to work with rate limits.

Result Format

No matter what you're searching for, the return value from the API follows a certain format. Here's a sample result from a query, which has been heavily edited to focus only on the parts you'll always see:

```
{
  "total_count": 824,
  "incomplete_results": false,
  "items": [
    {
      …
      "score": 3.357718
    }
  ]
}
```

Starting from the top: the `total_count` field represents the total number of search results that turned up from this query. It's not uncommon for a fairly specific search to turn up thousands of results – remember, there are millions of repositories on GitHub. By default, only the first 30 are returned, but you can customize this with `page` and `per_page` query parameters in the url. For example, a GET request to `search/repositories?q=foobar&page=2&page_size=45` will return 45 items, starting with the 46th result. Page sizes are generally limited to 100.

The `incomplete_results` field refers to a computational limit placed on the search API. If your search takes too long, the GitHub API will stop it partway through executing, return the results that did complete, and set this flag to `true`. For most queries this won't be a problem, and the `total_count` will represent all the results from the search, but if your query is complicated, you might only get a partial result set.

Search results are returned in the `items` array, and each item always has a `score` field. This field is numeric, but it's only a relative measure of how well a result matches the query, and is used for the default sort order – highest score first. If you do pay attention to it, remember it only has meaning when compared to other results from the same query.

Search Operators and Qualifiers

Of course, it's always better if you can avoid pagination altogether, or at least get the best results in the first page. Qualifiers and operators can help narrow your search results to fewer pages, hopefully allowing the right result to float to the top.

All searches are done through a search query, which is encoded and passed in the URL as the `q` parameter. Most of the query will be free text, but the API also supports some powerful syntax, such as these forms:

- `x AND y`, as well as `OR` and `NOT`
- `user:<name>`, where *name* is a user or organization
- `repo:<name>`
- `language:<name>`
- `created:<date(s)>`
- `extension:<pattern>` matches file extensions (like "py" or "ini")

Numerical values and dates can have ranges:

- `2015-02-01` will match only the given date
- `<2015-02-01` will match any date previous to the one given
- `2015-02-01..2015-03-01` will match dates within the given range, including the end points

There are many other options besides. Check out *https://github.com/search/advanced* for a UI that can help you construct a query.

Sorting

If search query operators can't narrow down a result set to just the most important items, perhaps sorting them can. Search results are returned in a definite order, never at random. The default order is "best match," which sorts your results based on their search score, best score first. If you want to override this, you can pass `stars`, `forks`, or `updated` in the `sort` query parameter, as in `search/repositories?q=foobar&sort=stars`.

You can also reverse the sort order using the `order` parameter, like `search/repositories?q=foobar&sort=stars&order=desc`. The default is `desc` ("descending"), but `asc` is also accepted, and will reverse the order.

Search APIs in Detail

Now that we've covered how all these APIs behave the same, let's discuss their specifics. The search API is compartmentalized into four categories: repositories, code, issues, and users. The basic mechanism is the same for all four: send a GET request to the endpoint, and provide a URL-encoded search term as the `q` parameter. We'll show an abridged response from each of the four, along with some discussion of what to expect.

Repository Search

The `search/repositories` endpoint looks in the repository metadata to match your query. This includes the project's name and description by default, though you can also search the README file by specifying `in:readme` in the query. Other qualifiers are documented at *https://developer.github.com/v3/search/#search-repositories*.

Its response looks something like this:

```
{
  "total_count": 824,
  "incomplete_results": false,
  "items": [
    {
      "id": 10869370,
      "name": "foobar",
      "full_name": "iwhitcomb/foobar",
      "owner": {
        "login": "iwhitcomb",
        "id": 887528,
        "avatar_url": "https://avatars.githubusercontent.com/u/887528?v=3",
        …
      },
      "private": false,
      "html_url": "https://github.com/iwhitcomb/foobar",
      "description": "Drupal 8 Module Example",
      "fork": false,
      …
```

```
      "score": 59.32314
    },
    …
  ]
}
```

Each item in `items` looks a lot like the result of a query to the repositories API. All sorts of useful information is included, such as a URL to the UI for this repository (`html_url`), the owner's avatar (`owner.avatar_url`), and a URL suitable for cloning the repository using Git (`git_url`).

Code Search

The `search/code` endpoint is for searching the contents of a repository. You can try matching the contents of the files themselves, or their paths (using `in:path`). (For complete documentation on the other available qualifiers, check out *https://developer.github.com/v3/search/#search-code*.)

This API is subject to several limits that don't affect the other search endpoints, because of the sheer amount of data the server must sort through to find matches. First, it requires that you provide a free-text search term; specifying a query with *only* operators (like `language:python`) is valid with other APIs, but not here. Second, any wildcard characters in the query will be ignored. Third, files above a certain size will not be searched. Fourth, it only searches the default branch of any given project, which is usually `master`. Fifth, and possibly most importantly, you *must* specify a repository owner using the `user:<name>` qualifier; you cannot search all repositories with one query.

The JSON returned looks something like this:

```
{
  "total_count": 9246,
  "incomplete_results": false,
  "items": [
    {
      "name": "migrated_0000.js",
      "path": "test/fixtures/ES6/class/migrated_0000.js",
      "sha": "37bdd2221a71b58576da9d3c2dc0ef0998263652",
      "url": "…",
      "git_url": "…",
      "html_url": "…",
      "repository": {
        "id": 2833537,
        "name": "esprima",
        "full_name": "jquery/esprima",
        "owner": {
          "login": "jquery",
          "id": 70142,
          "avatar_url": "https://avatars.githubusercontent.com/u/70142?v=3",
          …
```

```
        },
        "private": false,
        …
      },
      "score": 2.3529532
    },
    …
  ]
}
```

Each item has some data about the file that turned up, including its name and URLs for a couple of representations of it. Then there's the blob of data about its repository, followed by a score, which is used for the default "best match" sorting.

Issue Search

Repositories contain more than just code. The `search/issues` endpoint looks for matches in the issues and pull requests attached to a project. This endpoint responds to a wide variety of search qualifiers, such as:

- `type` – either "pr" for pull requests, or "issue" for issues (the default is both)
- `team` - match issues whose discussions mention a specific team (only works for organizations you belong to)
- `no` - match issues that are missing a piece of data (as in "no:label")

There are many more; see *https://developer.github.com/v3/search/#search-issues* for complete documentation.

The result of a call to this endpoint looks like this:

```
{
  "total_count": 1278397,
  "incomplete_results": false,
  "items": [
    {
      "url": "…",
      "labels_url": "…",
      "comments_url": "…",
      "events_url": "…",
      "html_url": "…",
      "id": 69671218,
      "number": 1,
      "title": "Classes",
      "user": {
        "login": "reubeningber",
        "id": 2552792,
        "avatar_url": "…",
        …
      },
```

```
        "labels": [
          …
        ],
        "state": "open",
        "locked": false,
        "assignee": null,
        "milestone": null,
        "comments": 0,
        "created_at": "2015-04-20T20:18:56Z",
        "updated_at": "2015-04-20T20:18:56Z",
        "closed_at": null,
        "body": "There should be an option to add classes to the ul and li being generated. ",
        "score": 22.575937
      },
    ]
}
```


Again, each item in the list looks like the result of a call to the issues API. There are a lot of useful bits of data here, such as the issue's title (`title`), labels (`labels`), and links to information about the pull-request data (`pull_request.url`), which won't be present if the result isn't a pull request.

User Search

All the other search APIs are centered around repositories, but this endpoint searches a different namespace: GitHub users. By default, only a user's login name and public email address are searched; the `in` qualifier can extend this to include the user's full name as well, with `in:fullname,login,email`. There are several other useful qualifiers available; see *https://developer.github.com/v3/search/#search-users* for complete documentation.

Querying the `search/users` endpoint gives you this kind of response:


```
{
  "total_count": 26873,
  "incomplete_results": false,
  "items": [
    {
      "login": "ben",
      "id": 39902,
      "avatar_url": "…",
      "gravatar_id": "",
      "url": "…",
      "html_url": "…",
      …
      "score": 98.24275
    },
    {
      "login": "bengottlieb",
      "id": 53162,
```

```
            "avatar_url": "…",
            "gravatar_id": "",
            "url": "…",
            "html_url": "…",
            …
            "score": 35.834213
          },
        ]
    }
```

The list of items in this case look like the results from a query of the `users/<name>` endpoint. Useful items here are the user's avatar (`avatar_url`), several links to other API endpoints (`repos_url`, `url`), and the type of result (user or organization, in `type`).

Our example application

Now that we know a bit about how this API behaves, let's do something useful with it.

Imagine your development team uses GitHub to store their Git repositories, and that there are lots of little repositories for parts of the application that work together at runtime. This kind of situation ends up being fairly difficult to work with for your non-technical colleagues; if they want to report an issue, they don't know where to go, and they don't know how to find issues that already exist.

Search can make this possible, but doing a search across an entire organization's repositories involves using the `user:<organization>` operator, which is obtusely named, and kind of scary for non-programmers.

The Search API can make this a bit easier. Let's make a GUI application with just a single search box, which makes it dead simple for a non-technical user to search all the issues in all the repositories in a single organization. It'll end up looking a bit like Figure 4-1.

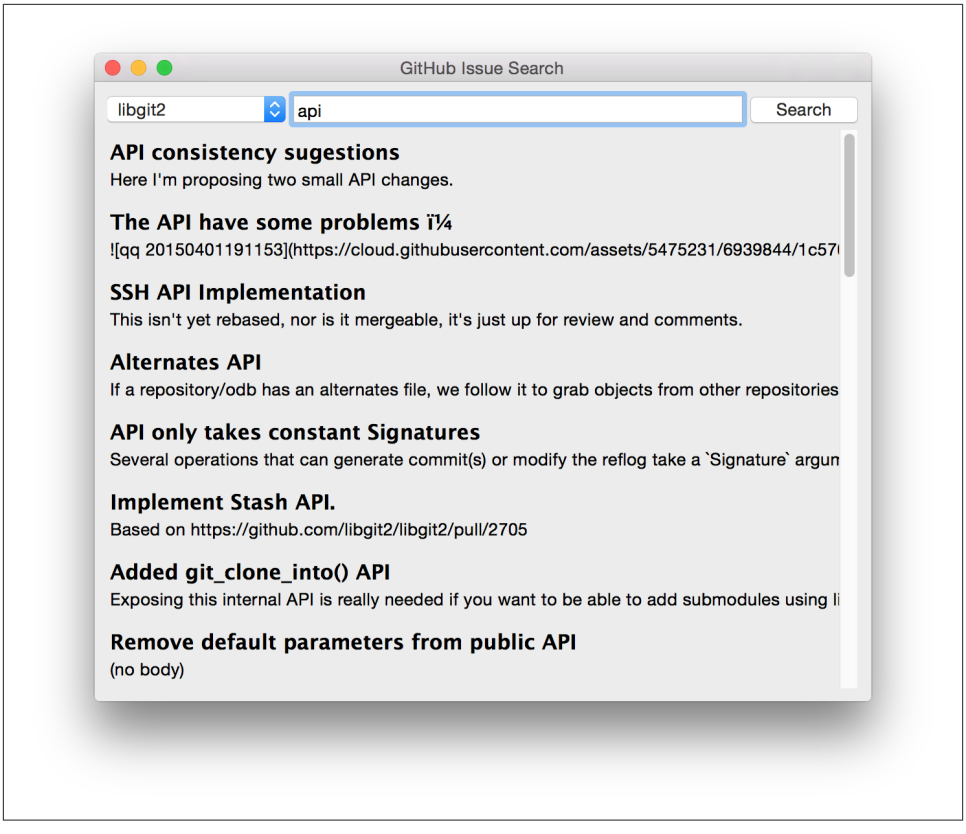


Figure 4-1. GitHub search application

User flow

That's the overall goal, but let's dig in to more detail about how the user experiences the application.

The first thing we'll do is require the user to log in with GitHub credentials. Why? Partly because the search API is throttled pretty aggressively, and the rate limits are higher with authenticated access. But also because our user is going to need the ability to search issues in private repositories. To make this easier, our program will try to get GitHub credentials from Git's credential store, but it'll fall back to a login form, which looks like Figure 4-2.

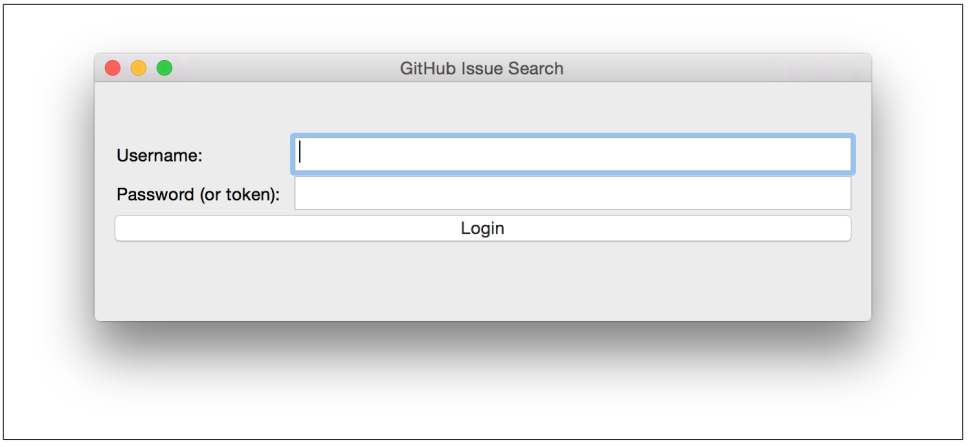


Figure 4-2. Login UI

Once the user logs in, they'll be shown a search box. Typing in a search query and hitting enter will result in a scrollable list of search results, with titles and the first line of the description. Clicking on a search result opens the issue in the user's browser.

That's about it. This application only has two main screens from the user's point of view. It's a simple, focused tool to solve a very tightly-defined problem, so the code shouldn't be too hard.

Python

Now that we know how the program should act, let's decide how it should *work*.

We'll use Python for our implementation language, for several reasons. First, because we haven't yet seen it in this book, and we like to expose you to a wide variety of languages. One of our goals is to help the reader explore technologies they might not have seen before.

Secondly, there's a library for building GUI applications that run without modification on Mac OS X, Linux, and Windows. Surprisingly, this is fairly unique feature among modern high-level programming languages. If you want this capability elsewhere, you usually have to use a high-complexity framework, a lower-level language like C++, or both.

Thirdly, this will help make it easy to distribute. Python has a package available which There exists a Python package which bundles an entire Python program and all of its dependencies into a single file (or `.app` bundle on OS X). So giving this program to a colleague is as easy as emailing her a ZIP file.

Let's take a quick look at the libraries we'll be using in our application's code. We'll see them in action later on, but a quick overview will help you understand what each one is trying to do. As is unfortunately typical with Python development, installation methods vary from package to package, so we'll also tell you how to get each one onto your machine.

AGitHub

The first thing we should mention is the library we'll use to talk to the GitHub API, which is called `agithub`. `agithub` implements is a very thin layer that converts GitHub's REST API into method calls on objects, resulting in delightfully readable code.

`agithub` can be found at *https://github.com/jpaugh/agithub*, and the "installation" is simply to download a copy of the `agithub.py` source file and place it alongside your project files.

WxPython

WxPython is how we'll create the graphical interface for our application. It's an object-oriented Python layer over the top of a toolkit called WxWidgets, which is itself a common-code adapter for native UI toolkits. WxWidgets supports Linux, Mac, and Windows operating systems with native controls, so you can access all of those platforms with the same Python code.

Information about the WxPython project can be found at *http://www.wxpython.org*, and you'll find a download link for your platform on the left-hand side of the page. The next version of WxPython (code-named "Phoenix"), will be installable via PIP, but as of this writing Phoenix is still pre-release software, so it's probably safer to use the stable version.

WxPython is the reason we use Python 2.7 for this example. Support for Python 3 is planned for the upcoming Phoenix release, but as of this writing, the stable versions are Python 2 only. Most of the code below is written in a "polyglot" fashion, so you shouldn't run into any trouble running it under Python 3 if Phoenix has arrived by the time you read this.

PyInstaller

PyInstaller will be our distribution tool. Its main function is to read your Python code, analyze it to discover all its dependencies, then collect all these files (including the Python interpreter) and put them in one directory. It can even wrap all of that up in a single package that, when double-clicked, runs your program. It does all this without

needing much input from you, and there are only a few configuration options. If you've written GUI applications before, you'll know how hard each of these problems are.

For information on this project, you can visit *http://pythonhosted.org/PyInstaller*. You can install it using Python's package manager, by running `pip install pyinstaller`.

The Code

Alright, now you have an idea of which parts of the Python ecosystem will be helping us on our journey. Let's get started looking at the code that brings them all together. We'll start with this skeleton file:

```
#!/usr/bin/env python # ❶

import os, subprocess
import wx
from agithub import Github # ❷

class SearchFrame(wx.Frame): # ❸
    pass

if __name__ == '__main__': # ❹
    app = wx.App() ❺
    SearchFrame(None)
    app.MainLoop()
```

If you run this program, you should get an empty window, which is always a hopeful start to any project. Let's take a look at a few key things:

❶ The "shebang" specifies that this is a Python 2.7 program

❷ Here we import our handy libraries. We import WxPython (`wx`) whole cloth, but with `agithub` we only need the `Github` (note the capitalization) class. `os` and `subprocess` come from the Python standard library.

❸ This is the class for our main window. We'll walk through the particulars later on when we discuss the real implementation.

❹ In Python, you create the main entry point of an application using this syntax.

❺ And this is how you write a "main" function in WxPython. We instantiate an `App` instance, create an instance of our top-level frame, and run the app's main loop.

Git credential helper

That's how most of the UI code is going to be structured, but before we go any further, we should define a function to help us get the user's GitHub credentials. We'll be cheating a bit, by asking Git if it has the user's login and password.

We'll leverage the `git credential fill` command. This is used internally by Git to avoid having to ask the user for their GitHub password every time they interact with a GitHub remote. The way it works is by accepting all the known facts about a connection through `stdin`, as text lines in the format "<key>=<value>". Once the caller has supplied all the facts it knows, it can close the stdin stream (or supply an empty line), and Git will respond with all the facts *it* knows about this connection. With any luck, this will include the user's login and password. The whole interaction looks a bit like this:

```
$ echo "host=github.com" | git credential fill ❶
host=github.com
username=ben ❷
password=(redacted)
```

❶ This passes a single line to `git credential` and closes `stdin`, which Git will recognize as the end of input.

❷ Git responds with all the facts it knows about the connection. This includes the input values, as well as the username and password if Git knows them.

One other thing that you should know about `git-credential` is that by default, if it doesn't know anything about the host, it'll ask the user at the terminal. That's bad for a GUI app, so we're going to be disabling that feature through the use of the `GIT_ASKPASS` environment variable.

Here's what our helper looks like:

```
GITHUB_HOST = 'github.com'
def git_credentials():
    os.environ['GIT_ASKPASS'] = 'true' ❶
    p = subprocess.Popen(['git', 'credential', 'fill'],
                         stdout=subprocess.PIPE,
                         stdin=subprocess.PIPE) ❷
    stdout,_ = p.communicate('host={}\n\n'.format(GITHUB_HOST)) ❸

    creds = {}
    for line in stdout.split('\n')[:-1]: ❹
        k,v = line.split('=')
        creds[k] = v
    return creds ❺
```

❶ Here we set `GIT_ASKPASS` to the string `'true'`, which is a UNIX program that always returns 0, which will cause `git-credential` to stop trying to get credentials when it gets to the "as the user" stage.

❷ `subprocess.Popen` is the way you use a program with stdin and stdout in Python. The first argument is the `argv` of the new program, and we also specify that we want stdin and stdout to be captured.

❸ `p.communicate` does the work of writing to stdin and returning the contents of stdout. It also returns the contents of stderr, which we ignore in this program.

❹ Here we process the stdout contents by splitting each line at the = character, and slurping it into a dictionary.

So the return value from this call should be a dictionary with `'username'` and `'password'` values. Handy!

Windowing and interface

Okay, so now we have something that can help us skip a login screen, but we don't have a way of showing that login screen to the user. Let's get closer to that goal by filling in the main frame's implementation:

```
class SearchFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        kwargs.setdefault('size', (600,500))
        wx.Frame.__init__(self, *args, **kwargs)

        self.credentials = {}
        self.orgs = []

        self.create_controls()
        self.do_layout()

        # Try to pre-load credentials from Git's cache
        self.credentials = git_credentials()
        if self.test_credentials():
            self.switch_to_search_panel()

        self.SetTitle('GitHub Issue Search')
        self.Show()
```

The `__init__` method is the constructor, so this is where we start when the main function calls `SearchFrame()`. Here's what's happening at a high level – we'll dig into the details in a bit:

1. Set up some layout dimensions and pass to the parent class's constructor
2. Create the UI controls
3. Retrieve the credentials from the user using the credential helper we described earlier
4. Change the title and display the application to the user

Before we get to *how* all those things are done, let's step back a bit and talk about this class's job. It's responsible for maintaining the top-level "frame" (a window with a title bar, a menu, and so on), and deciding what's displayed in that frame. In this case, we

want to show a login UI first, and when we get valid credentials (either from Git or the user), we'll switch to a searching UI.

Alright, enough background. Let's walk through the code for getting and checking credentials.

```
    def login_accepted(self, username, password):
        self.credentials['username'] = username
        self.credentials['password'] = password
        if self.test_credentials():
            self.switch_to_search_panel()

    def test_credentials(self):
        if any(k not in self.credentials for k in ['username', 'password']):
            return False
        g = Github(self.credentials['username'], self.credentials['password'])
        status,data = g.user.orgs.get()
        if status != 200:
            print('bad credentials in store')
            return False
        self.orgs = [o['login'] for o in data]
        return True

    def switch_to_search_panel(self):
        self.login_panel.Destroy()
        self.search_panel = SearchPanel(self,
                                        orgs=self.orgs,
                                        credentials=self.credentials)
        self.sizer.Add(self.search_panel, 1, flag=wx.EXPAND | wx.ALL, border=10)
        self.sizer.Layout()
```

Each of these three methods comes in at a different point during our program's execution. If our credentials are coming from Git, we proceed straight to `test_credentials`; if they're coming from the login panel (see below), they go through the `login_accepted` callback first, which then calls `test_credentials`.

Either way, what we do is try to fetch a list of the user's organizations, to see if they work. Here you can see the usage pattern for `agithub` – the URL path is mapped to object-property notation on an instance of the `Github` class, and the HTTP verb is mapped to a method call. The return values are a status code and the data, which has been decoded into a dictionary object. If it fails — meaning the returned status is not `200` — we send the user to the login panel. If it succeeds, we call `switch_to_search_panel`.

We're doing a synchronous network call on the UI thread. This is usually a bad idea, because the UI will become unresponsive until the network call completes. Ideally we'd move this out onto another thread, and get the return value with a message, but for this simple example (and use case), it'll do.

The last method handles the UI switch. The login panel is referenced by two things: the `SearchFrame` instance (the parent window), and the sizer that's controlling its layout. Fortunately, calling the `Destroy()` method cleans both of those up, so we can then create the `SearchPanel` instance and add it to our sizer. Doing this requires a specific call to the sizer's `Layout()` method; otherwise the sizer won't know that it needs to adjust the position and size of the new panel.

```
    def create_controls(self):
        # Set up a menu. This is mainly for "Cmd-Q" behavior on OSX
        filemenu = wx.Menu()
        filemenu.Append(wx.ID_EXIT, '&Exit')
        menuBar = wx.MenuBar()
        menuBar.Append(filemenu, '&File')
        self.SetMenuBar(menuBar)

        # Start with a login UI
        self.login_panel = LoginPanel(self, onlogin=self.login_accepted)

    def do_layout(self):
        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.login_panel, 1, flag=wx.EXPAND | wx.ALL, border=10)
        self.SetSizer(self.sizer)
```

`create_controls` is fairly straightforward. It instantiates a menu that only contains File>Exit, and a login panel, whose implementation we'll cover a bit later on. Note that when we create a visible control, we pass `self` as the first parameter to the constructor. That's because the `SearchFrame` instance we're constructing is the parent window of that control.

`do_layout` uses a WxWidgets feature called "sizers" to do some automated layout. Sizers are a complex topic, but here's all you need to know about this snippet:

- A `BoxSizer` stacks widgets in a single direction, in this case vertically.
- The second parameter to `sizer.Add` is a scaling factor. If it's zero, the widget you're adding will always stay the same size if the parent window resizes; if it's anything else, all the things the sizer is controlling will adjust to fill their container. There's only one control in this sizer, but we still want it to take up the full area of the window, so we pass `1`.
- The `border` parameter tells the sizer how much area to leave around the widget as padding.
- The `wx.EXPAND` flag tells the sizer that we want the widget to expand in the direction the sizer isn't stacking. In this case, we're stacking vertically, but we also want this widget to expand horizontally.
- The `wx.ALL` flag specifies which edges of the widget should have the border area.

That's it! Aside from managing a couple of fields, most of this code is managing the UI, which is almost exactly what we'd want from a UI class. Let's write the first of the two panels that we swap in and out.

GitHub login

The `LoginPanel` class is similar in structure to the `SearchFrame` class, with a couple of key differences, which we'll describe after the wall of code.

```
class LoginPanel(wx.Panel):
    def __init__(self, *args, **kwargs):
        self.callback = kwargs.pop('onlogin', None)
        wx.Panel.__init__(self, *args, **kwargs)

        self.create_controls()
        self.do_layout()

    def create_controls(self):
        self.userLabel = wx.StaticText(self, label='Username:')
        self.userBox = wx.TextCtrl(self, style=wx.TE_PROCESS_ENTER)
        self.passLabel = wx.StaticText(self, label='Password (or token):')
        self.passBox = wx.TextCtrl(self, style=wx.TE_PROCESS_ENTER)
        self.login = wx.Button(self, label='Login')
        self.error = wx.StaticText(self, label='')
        self.error.SetForegroundColour((200,0,0))

        # Bind events
        self.login.Bind(wx.EVT_BUTTON, self.do_login)
        self.userBox.Bind(wx.EVT_TEXT_ENTER, self.do_login)
        self.passBox.Bind(wx.EVT_TEXT_ENTER, self.do_login)

    def do_layout(self):
        # Grid arrangement for controls
        grid = wx.GridBagSizer(3,3)
        grid.Add(self.userLabel, pos=(0,0),
                 flag=wx.TOP | wx.LEFT | wx.BOTTOM, border=5)
        grid.Add(self.userBox, pos=(0,1),
                 flag=wx.EXPAND | wx.LEFT | wx.RIGHT, border=5)
        grid.Add(self.passLabel, pos=(1,0),
                 flag=wx.TOP | wx.LEFT | wx.BOTTOM, border=5)
        grid.Add(self.passBox, pos=(1,1),
                 flag=wx.EXPAND | wx.LEFT | wx.RIGHT, border=5)
        grid.Add(self.login, pos=(2,0), span=(1,2),
                 flag=wx.EXPAND | wx.LEFT | wx.RIGHT, border=5)
        grid.Add(self.error, pos=(3,0), span=(1,2),
                 flag=wx.EXPAND | wx.LEFT | wx.RIGHT, border=5)
        grid.AddGrowableCol(1)

        # Center the grid vertically
        vbox = wx.BoxSizer(wx.VERTICAL)
        vbox.Add((0,0), 1)
```

```
        vbox.Add(grid, 0, wx.EXPAND)
        vbox.Add((0,0), 2)
        self.SetSizer(vbox)

    def do_login(self, _):
        u = self.userBox.GetValue()
        p = self.passBox.GetValue()
        g = Github(u, p)
        status,data = g.issues.get()
        if status != 200:
            self.error.SetLabel('ERROR: ' + data['message'])
        elif callable(self.callback):
            self.callback(u, p)
```

There's some structure that's similar to above. We'll start with the constructor.

Recall that this panel is created with a keyword argument in the `SearchFrame` class, like `LoginPanel(self, onlogin=self.login_accepted)`. In the constructor definition, we pull that callback out and store it for later. Afterward, we just call the two other construction functions and return.

`create_controls` has more to it than `SearchFrame`'s version, because this panel has more controls. Every static-text, text-input, and button control gets its own line of code. The `wx.TE_PROCESS_ENTER` style tells the library that we want an event to be triggered if the user presses the enter key while the cursor is inside that text box.

The next block binds control events to method calls. Every event in WxPython will call the handler with a single argument, an object which contains information about the event. That means we can use the same event handler for any number of different kinds of events, so we do – the `ENTER` handlers for both text boxes and the `BUTTON` handler for the button all go through `self.do_login`.

`do_layout` uses a different kind of sizer – a `GridBagSizer`. Again, the topic of sizers is *way* outside the scope of this chapter, but just know that this kind arranges things in a grid, and you can allow some of the rows or columns to stretch to fill the container. Here we drop all of the controls into their positions with the `pos=(r,c)` notation (here "rows" come first, which isn't like most coordinate systems), and cause one control to span two columns with the `span` parameter. The `flags` and `border` parameters mostly mean the same thins as before, and the `AddGrowableCol` function tells the layout engine which parts of the grid should be allowed to stretch.

Then we do something curious: we put the `GridBagSizer` *into another sizer*. Sizer nesting is a powerful feature, and allows almost any window layout to be possible — although perhaps not easy or simple. The vertical box sizer also contains some bare tuples; this special form is called "adding a spacer." In this case, we sandwich the sizer with all the controls between two spacers with different weights, making it float about a third of the way down the window. The effect is like Figure 4-3.

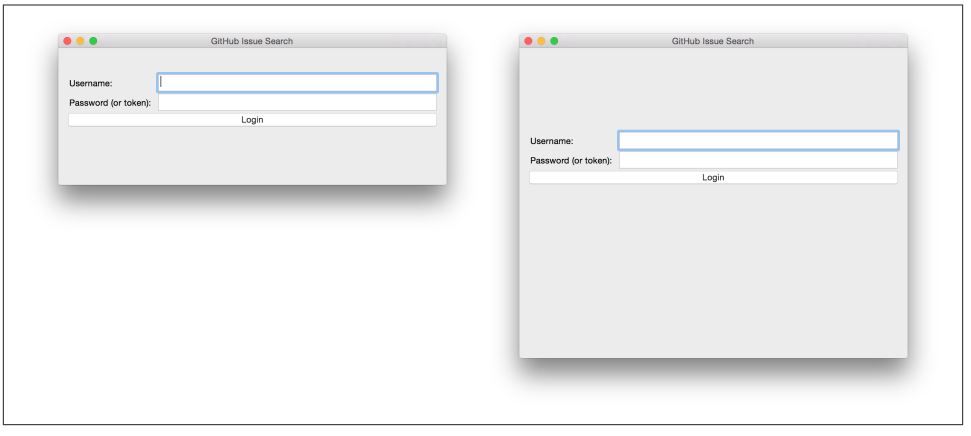


Figure 4-3. Resizing behavior of login UI

Then comes the `do_login` method, which tests out the given credentials, and if they work, passes them back through the callback set at construction time. If they don't work, it sets the text of a label, whose foreground color has been set to a nice, alarming shade of red.

GitHub search

Once the user has successfully logged in, we destroy the `LoginPanel` instance and show the `SearchPanel`.

```
class SearchPanel(wx.Panel):
    def __init__(self, *args, **kwargs):
        self.orgs = kwargs.pop('orgs', [])
        self.credentials = kwargs.pop('credentials', {}) ❶
        wx.Panel.__init__(self, *args, **kwargs)

        self.create_controls()
        self.do_layout()

    def create_controls(self):
        self.results_panel = None
        self.orgChoice = wx.Choice(self, choices=self.orgs, style=wx.CB_SORT)
        self.searchTerm = wx.TextCtrl(self, style=wx.TE_PROCESS_ENTER)
        self.searchTerm.SetFocus()
        self.searchButton = wx.Button(self, label="Search")

        # Bind events
        self.searchButton.Bind(wx.EVT_BUTTON, self.do_search)
        self.searchTerm.Bind(wx.EVT_TEXT_ENTER, self.do_search)

    def do_layout(self):
```

```
        # Arrange choice, query box, and button horizontally
        hbox = wx.BoxSizer(wx.HORIZONTAL)
        hbox.Add(self.orgChoice, 0, wx.EXPAND)
        hbox.Add(self.searchTerm, 1, wx.EXPAND | wx.LEFT, 5)
        hbox.Add(self.searchButton, 0, wx.EXPAND | wx.LEFT, 5)

        # Dock everything to the top, leaving room for the results
        self.vbox = wx.BoxSizer(wx.VERTICAL)
        self.vbox.Add(hbox, 0, wx.EXPAND) ❷
        self.SetSizer(self.vbox)

    def do_search(self, event):
        term = self.searchTerm.GetValue()
        org = self.orgChoice.GetString(self.orgChoice.GetCurrentSelection())
        g = Github(self.credentials['username'], self.credentials['password'])
        code,data = g.search.issues.get(q="user:{} {}".format(org, term)) ❸
        if code != 200:
            self.display_error(code, data)
        else:
            self.display_results(data['items'])

    def display_results(self, results): ❹
        if self.results_panel:
            self.results_panel.Destroy()
        self.results_panel = SearchResultsPanel(self, -1, results=results)
        self.vbox.Add(self.results_panel, 1, wx.EXPAND | wx.TOP, 5)
        self.vbox.Layout()

    def display_error(self, code, data): ❺
        if self.results_panel:
            self.results_panel.Destroy()
        if 'errors' in data:
            str = ''.join('\n\n{}'.format(e['message']) for e in data['errors'])
        else:
            str = data['message']
        self.results_panel = wx.StaticText(self, label=str)
        self.results_panel.SetForegroundColour((200,0,0))
        self.vbox.Add(self.results_panel, 1, wx.EXPAND | wx.TOP, 5)
        self.vbox.Layout()
        width = self.results_panel.GetSize().x
        self.results_panel.Wrap(width)
```

There's quite a bit here, but some of it is familiar. We'll skip the usual walkthrough, to point out a couple of interesting features.

❶ When creating the panel, we pass in the user's credentials and list of organizations.

❷ When we add the search bar to the sizer, we use `0` as a scale factor. This means that it shouldn't exand to fit the available size, but keep its own size instead, to leave room to add a results panel later on.

❸ Here's where the actual search is being done.

❹ We pass the search results into another class, then add it to the main sizer with parameters to fill the remaining available space.

❺ If an error is returned from the search call instead, we display it here. There's some code to adjust the wrap width of the text, based on the laid-out width of the control. This isn't a great approach; doing it better is left as an exercise for the reader.

Again, there's a fair amount of code here, but most of it should look familiar.

Displaying results

So now we have our login panel, and a way for the user to enter a search query, but no way to display results. Let's fix that.

Whenever search results are retrieved, we create a new instance of `SearchResultsPanel`, which then creates a series of `SearchResult` instances. Let's look at both of them together:

```
class SearchResultsPanel(wx.ScrolledWindow): ❶
    def __init__(self, *args, **kwargs):
        results = kwargs.pop('results', [])
        wx.PyScrolledWindow.__init__(self, *args, **kwargs)

        # Layout search result controls inside scrollable area
        vbox = wx.BoxSizer(wx.VERTICAL)
        if not results:
            vbox.Add(wx.StaticText(self, label="(no results)"), 0, wx.EXPAND)
        for r in results:
            vbox.Add(SearchResult(self, result=r),
                     flag=wx.TOP | wx.BOTTOM, border=8)
        self.SetSizer(vbox)
        self.SetScrollbars(0, 4, 0, 0)

class SearchResult(wx.Panel):
    def __init__(self, *args, **kwargs):
        self.result = kwargs.pop('result', {})
        wx.Panel.__init__(self, *args, **kwargs)

        self.create_controls()
        self.do_layout()

    def create_controls(self): ❷
        titlestr = self.result['title']
        if self.result['state'] != 'open':
            titlestr += ' ({})'.format(self.result['state'])
        textstr = self.first_line(self.result['body'])
        self.title = wx.StaticText(self, label=titlestr)
        self.text = wx.StaticText(self, label=textstr)
```

```
        # Adjust the title font
        titleFont = wx.Font(16, wx.FONTFAMILY_DEFAULT,
                            wx.FONTSTYLE_NORMAL, wx.FONTWEIGHT_BOLD)
        self.title.SetFont(titleFont)

        # Bind click and hover events on this whole control ❸
        self.Bind(wx.EVT_LEFT_UP, self.on_click)
        self.Bind(wx.EVT_ENTER_WINDOW, self.enter)
        self.Bind(wx.EVT_LEAVE_WINDOW, self.leave)

    def do_layout(self):
        vbox = wx.BoxSizer(wx.VERTICAL)
        vbox.Add(self.title, flag=wx.EXPAND | wx.BOTTOM, border=2)
        vbox.Add(self.text, flag=wx.EXPAND)
        self.SetSizer(vbox)

    def enter(self, _):
        self.title.SetForegroundColour(wx.BLUE)
        self.text.SetForegroundColour(wx.BLUE)

    def leave(self, _):
        self.title.SetForegroundColour(wx.BLACK)
        self.text.SetForegroundColour(wx.BLACK)

    def on_click(self, event): ❹
        import webbrowser
        webbrowser.open(self.result['html_url'])

    def first_line(self, body):
        return body.split('\n')[0].strip() or '(no body)'
```

❶ The containing panel is simple enough that it only consists of a constructor. This class's job is to contain the results, and present them in a scroll window.

❷ A `SearchResult` comprises two static text controls, which contain the issue's title and the first line of its body.

❸ We're binding the click handler for this entire panel, but also the mouse-enter and mouse-leave events, so we can make it behavior more like a link in a browser.

❹ Here's how you open the default browser to a URL in Python.

Overall, WxPython isn't so bad, once you get used to it. It lacks some facilities of newer frameworks, but there's nothing better for getting a basic cross-platform UI out the door quickly.

That's all of the code! If you've been following along, you can run this code file and do issue searches. However, our use case has a non-technical user runnnig this; let's see what can be done to make it easy for them.

Packaging

What we're not going to do is require anyone to install Python 2.7 and a bunch of packages. We'll use PyInstaller to bundle our application into something that's easy to distribute and run.

Let's assume you wrote all the code above into a file called `search.py`, and `agithub.py` is sitting in the same directory. Here's how to tell PyInstaller to generate a single application for you:

```
$ pyinstaller -w search.py
```

That's it! The `-w` flag tells PyInstaller to create a "windowed" build of your application, rather than the default console build. On OS X, this generates a `search.app` application bundle, and on Windows this generates a `search.exe` file. You can take either of these to a computer with no Python installed, and they'll run perfectly.

That's because PyInstaller has copied everything necessary for your program to run, from the Python interpreter on up, inside that file. The one I just generated is 67MB, which seems large for such a simple program, but that number is more reasonable when you consider what's inside the package.

Summary

Whew! This chapter was quite a journey. Let's take a breath, and look at what we've learned.

The main bulk of the code in this chapter had to do with defining a graphical interface. Code for this task is always pretty verbose, because of the sheer complexity of the task. With WxPython in your tool belt, however, you can now write GUI applications using Python, with code that's no harder to write than with other toolkits, and get the ability to run on every major platform for free.

We saw how to ask Git for credentials to a Git server using 'git credential`. This feature is quite capable, and includes the ability to write a custom credential storage back-end, but we at least saw a peek into how it works. Using this knowledge, you can piggy-back on your users' existing habits to avoid having to ask them for the same things over and over again.

We also saw a rather nice HTTP API abstraction with `agithub`. We authenticated and queried the issue-search API endpoint, using what looked like object-method notation. `agithub` is a great example of how a library package can be both future-proof and idiomatic – the library constructs a query URL by looking at the chain of properties and methods used in the call. This is a great jumping-off point for querying other REST APIs using the same pattern.

Finally, the main thrust of this chapter was using the GitHub search API. You've learned about its general behavior, the different categories of search, how to interpret and sort results, and ways of focusing a search to reduce the number of uninteresting results. Using this knowledge you should be able to find anything you're looking for on GitHub or GitHub Enterprise. You also know that the search UI on GitHub is just a thin layer over the search API, so the same tricks and techniques will serve you whether you're writing code or using a browser.

In the next chapter we will look at using DotNET with the Commit Status API.

CHAPTER 5

DotNet and the Commit Status API

At the risk of oversimplifying things too much, one way to look at a Git repository is as just a long series of commits. Each commit contains quite a bit of information: the contents of the source files, who created the commit and when, the author's comments on what changes the commit introduces, and so on. This is all good stuff, and works very well for Git's main use case: controlling the history of a software project.

GitHub's commit-status API adds another layer of metadata to a commit: what various services *say* about that commit. This capability primarily shows itself in the pull-request UI, as shown in Figure 5-1. Each commit in the pull request is annotated with a symbol indicating its status - a red "×" for failure or error, a green "✓" for success, or an amber "•" to indicate that a decision is in the process of being made. This feature also surfaces at the bottom of the pull-request; if the last commit in the branch is not marked as successful, you get a warning about merging the request.

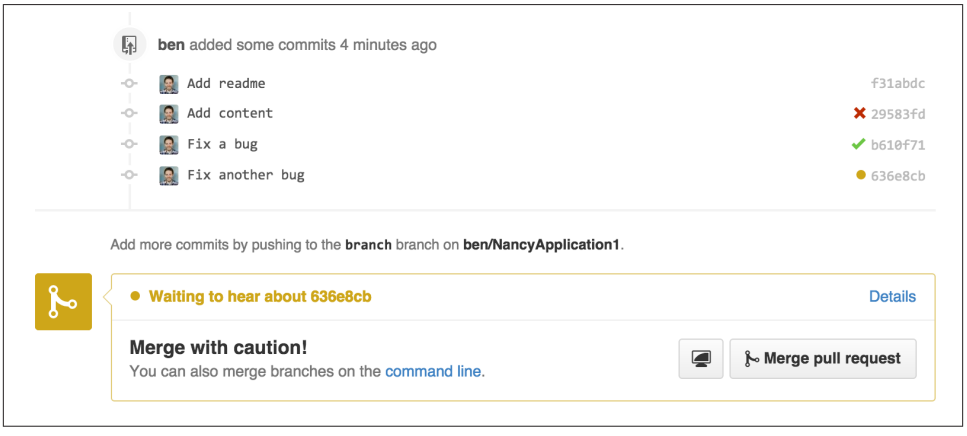


Figure 5-1. Commit status in the pull-request UI

The most obvious application for this feature is a continuous-integration service. A program like Jenkins will get a notification when new commits are pushed to a branch, run a build/test cycle using the new code, and post the results through the commit-status API. An application like this can even include a link back to the build results, so the user can find out which tests failed. This is a great way to bring together everything needed to make a decision about a proposal: what code has changed, what do people think about it, and does this change break anything? The answer to all of these questions is available on the same page: the pull-request conversation view.

Building and testing is only the beginning, though; the status of a commit can be used for other purposes as well. For example, open-source projects often have a license agreement that you must sign in order to submit a contribution. These are called "contributor license agreements," and usually contain language about licensing the contribution to the maintainers of the project. But it's tedious to manually check every incoming pull request to see if the author has signed the CLA, so a continuous-integration-style service can be used for this. CLAHub is one such example: it checks to see if all of the authors of the contained commits have signed the CLA, and marks the latest commit as "error" if not.

So now we know what the feature is, and what its intended use is. Let's take a look at how a program can interact with it.

The API

First, let's talk about access control. The commit status API exposes the need for OAuth as few others do. Making a repository private means you want complete control of what people or applications can access it. Naturally you trust GitHub's internal code to do the right thing with your data, but what about some random application from the Internet? OAuth gives you a way to grant private-repository access to an application *with limits* – the use of OAuth scopes allows an application to ask for a specific set of permissions, but it won't be able to do just any old thing with your data. Plus, these you're always in control of these permissions; you can revoke an application's access at any time.

The OAuth system includes the concept of scopes, which can be requested by and granted to an application, each of which allows a certain set of actions. The commit-status API requires the `repo:status` OAuth scope, which allows an application read and write access to **just** commit statuses; there is no access granted to the actual contents of the repository. This might seem strange: how can you judge the status of a commit without being able to inspect its contents? Just remember that this feature has use cases beyond continuous integration, and an application may not need full access to make a decision. For services that do need to be able to look at the repository contents, you can request the `repo` scope, which grants read *and* write access to the entire contents of a repository, including commit statuses. As of this writing, there's no way to request read-only access

to repositories, so if a service needs access to your data, you have to trust it with write access also.

Raw statuses

Now that we know how we get access to commit statuses, let's see what they look like. Commit statuses exist as atomic entities, and each commit can have a practically unlimited number of them (the actual number is in the thousands). You can query for existing statuses by doing a GET request to the API server at `/repos/<user>/<repo>/<ref>/statuses`, and it will return a list of them that looks like this:

```
[
  {
    "url": "https://api.github.com/repos/…",
    "id": 224503786,
    "state": "success",
    "description": "The Travis CI build passed",
    "target_url": "https://travis-ci.org/libgit2/libgit2/builds/63428108",
    "context": "continuous-integration/travis-ci/push",
    "created_at": "2015-05-21T03:11:02Z",
    "updated_at": "2015-05-21T03:11:02Z"
  },
  …
]
```

Most of this is self-explanatory, but a couple of fields need explaining. The `state` field can be "success," "failure," "error," or "pending," depending on the state of the service's decision. The `target_url` is a URL for the specific decision made for this commit (in this case a build/test log), which helps the user figure out why a particular decision was reached. And the `context` parameter is used for correlating multiple status updates to a single service; each application sets this according to its own rules, but any process that creates statuses should post the `pending` status and the result status using the same context value.

This API is useful for getting the raw data involved, but it gets complicated quickly. How do you decide if a given commit is "good?" What if there are 3 pending statuses, one success, another pending, two failures, and another success, in that order? The `context` field can help you correlate a single service's updates, and you can order them by `created_at` to see how each one turned out, but that's a lot of work. Fortunately, the API server can do it for you.

Combined status

If you instead do a GET to `/repos/<user>/<repo>/<ref>/status` (note that the last word is singular), you'll instead get a response that looks like this:

```
{
  "state": "success",
  "statuses": [
    {
      "url": "https://api.github.com/repos/…",
      …
    },
    { … }
  ],
  "sha": "6675aaba883952a1c1b28390866301ee5c281d37",
  "total_count": 2,
  "repository": { … },
  "commit_url": "https://api.github.com/repos/…",
  "url": "https://api.github.com/repos/…"
}
```

The `statuses` array is the result of the logic you'd probably write if you had to: it collapses the statuses by context, keeping only the last one. The `state` field contains an overall status that takes into account all of the contexts, providing a final value based on these rules:

- `failure` if any of the contexts posted a `failure` or `error` state
- 'pending` if any of the contexts' latest state is `pending` (or if there are no statuses)
- `success` if the latest status for every context is `success`

This is probably exactly what you want, but if you find that your use case calls for different rules, you can always use the `statuses` endpoint to get the raw data and calculate your own combined status.

Creating a status

Now obviously these statuses have to come from somewhere. This API also includes a facility for creating them. To do this, you simply make a POST request to `/repos/<user>/<repo>/statuses/<sha>`, and supply a JSON object for the fields you want to include with your status:

- `state` is required, and must be one of `pending`, `success`, `error`, or `failure`.
- `target_url` is a link to detailed information on the process of deciding what the state is or will be.
- `description` is a short string describing what the service is doing to make a decision.
- `context` is an application-specific string to allow the API to manage multiple services contributing to a single commit's status.

Notice how the last component in that URL is `<sha>`. While you can query for statuses or a combined status using a ref name (like `master`), creating a status requires you to know the full SHA-1 hash of the commit you want to annotate. This is to avoid race conditions: if you were targeting a ref, it may have moved between when your process started and when it finishes, but the SHA of a commit will never change.

Let's write an app

Alright, now that we know how to read and write statuses, let's put this API to work. In this chapter, we'll build a simple HTTP service that lets you create commit statuses for repositories you have access to, using the OAuth web flow for authorization. The system we'll build will be fairly limited in scope, but it's a great starting point to customize for your specific needs.

The language this time is C#, running on the CLR (Common Language Runtime). At one point in the history of computing this wouldn't have been a good choice for a book like this, since it was only available on Windows, the development tools cost quite a bit of money, and the language and libraries were fairly limited. However, with the advent of Mono (an open-source implementation of the .NET runtime), the open-sourcing of the CLR core, and the availability of free tools, C# is now a completely valid and rather nice option for open-source or hobby developers. Plus, it has a vibrant ecosystem of packages we can leverage to make our jobs easier.

Libraries

You'll be happy to know that we won't be writing an entire HTTP server from scratch in this chapter. There are a number of open-source packages that do this work for us, and in this project we'll be using Nancy. Nancy is a project that started as a CLR port of the Sinatra framework for Ruby (it takes its name from Frank Sinatra's daughter, Nancy). It's very capable, but also very succinct, as you'll see.

We also won't be directly implementing access to the GitHub API, because GitHub provides a CLR library for that. It's called octokit.net, and it does all the right things with regard to asynchrony and type safety. This is the same library used by the GitHub client for Windows, so it'll definitely do the job for our little application. It is, however, the source of a constraint on how we set up our example project: it requires a rather new version of the CLR (4.5) in order to function. If you want some guidance on how to avoid this pitfall and follow along, continue reading the next section. If you've worked with Nancy before, and have installed NuGet packages in the past, you might be able to skip to the section labeled "First steps" on page 89.

Following along

If you'd like to follow along with the code examples, here's how to set up a development environment with all the necessary elements. The process is different on Windows (using Visual Studio) and any other platforms (using Xamarin tools).

Visual Studio

If you're running Windows, you'll want to visit *https://www.visualstudio.com/* and download the Community edition of Visual Studio. The installer will present you with lots of options; for this example, we'll only need the "web developer" components, but feel free to check all the boxes that look interesting to you. (If you have access to a higher tier of Visual Studio, or already have it installed with the web-development packages, you're all set.)

In order to make things just a little smoother, you'll want to install a plugin: the Nancy project templates. Visit *https://visualstudiogallery.msdn.microsoft.com/* and search for "nancy.templates". As of this writing, there appears to be some difficulty with file formats, so when you download it, it comes as a ZIP file. If this has been resolved by the time you're reading this, simply double-click the file to install the templates; if not, you'll have to rename it to have a `.vsix` extension first.

The next step is to create a new project using one of the newly-installed templates. Go to "File>New Project…" and select "Visual C#>Web>Nancy Application with ASP.NET Hosting" from the template list (as shown in Figure 5-2. Make sure the path and name settings at the bottom are to your liking, and click OK.

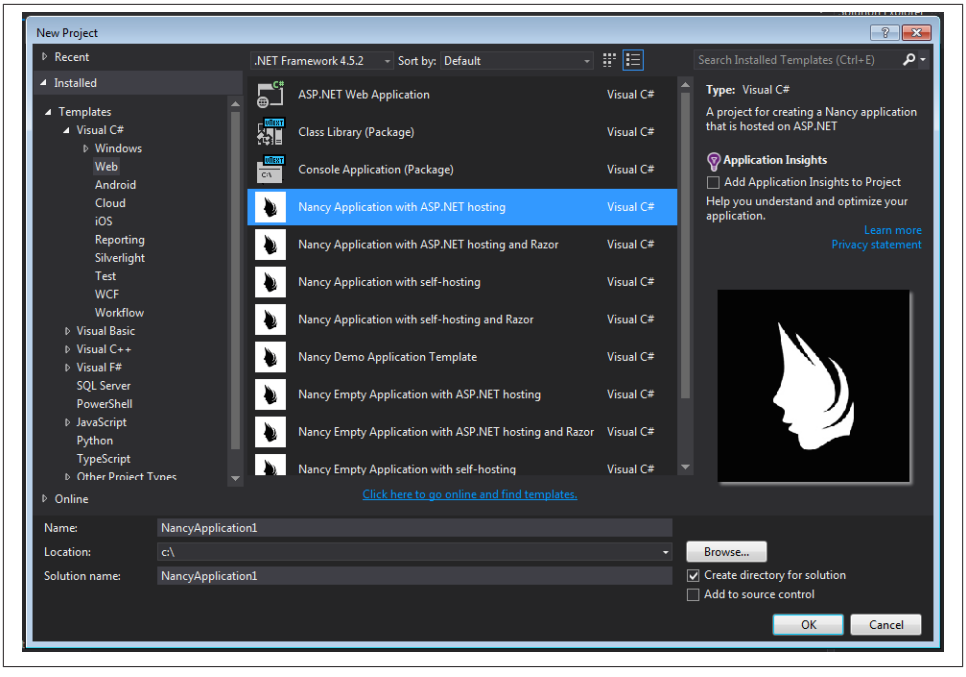


Figure 5-2. Creating a Nancy application in Visual Studio

Next, change the target CLR framework version to something that will work with Octokit. Right-click on the project's node in the Solution Explorer, and select "Properties." In the "Application" section, set Target Framework to be ".NET 4.5" (or later), and save. You may be prompted to re-load the solution.

The very last step is to add NuGet packages for Octokit and Nancy. Right-click on the project node in Solution Explorer, and select "Manage NuGet Packages…" Do a search for "Nancy", and upgrade it if necessary – there's a chance the Nancy project template specifies an out-of-date version. Then do a search for "Octokit," and install that. At this point, you should have an empty solution, configured and ready for our example code. To run it with debugging, go to "Debug>Start Debugging…," or hit F5. Visual Studio will start the server under a debugger, and open an IE instance on *http://localhost:12008/* (the port might be different), which should serve you the default Nancy "404 Not Found" page.

Xamarin Studio

On OS X and Linux, as of this writing the easiest way forward is to visit *http://www.monodevelop.com/* and install MonoDevelop. Mono is an open-source implementation of Microsoft's CLR specification, and MonoDevelop is a development environment that works much like Visual Studio, but is built on Mono, and is completely

open-source. If you try to download MonoDevelop on a Windows or OS X machine, you'll be prompted to install Xamarin Studio instead; this is a newer version of MonoDevelop with more capabilities, and will work just as well for these examples.

There are no Nancy-specific project templates for these IDEs, so you'll just start with an empty web project. Go to "File>New>Solution…", and choose "ASP.NET>Empty ASP.NET Project" from the template chooser, as shown in Figure 5-3.

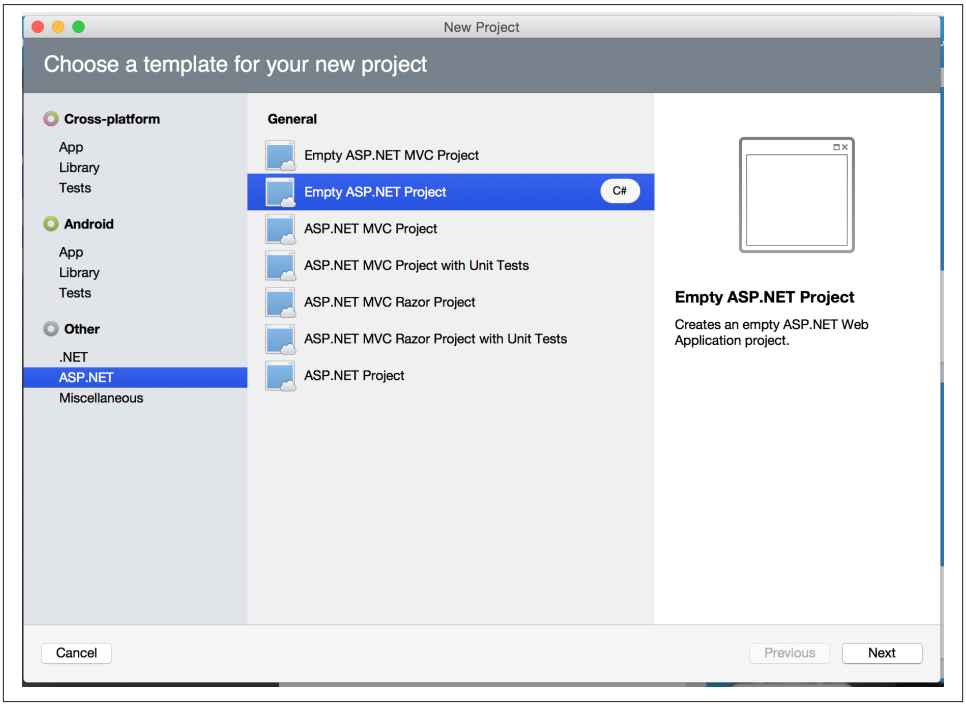


Figure 5-3. Creating an empty ASP.NET application in Xamarin Studio

The rest of the wizard steps are about the project name and location; feel free to name and locate this project however you like.

Next, update the target framework setting. Control- or right-click on the node in the solution explorer that corresponds with your project (*not* your solution), and select "Options" from the menu. Under "Build>General," set the Target Framework to "Mono / .NET 4.5" (or later) and click OK.

Lastly, install the Nancy and Octokit NuGet packages. Go to "Project>Add NuGet Packages…" in the menu to open the package manager. Search for Nancy, check the box next to it, search for Octokit, check its box, and click "Add Packages" at the bottom right. Once the process is complete, your project is ready for our example code. To run it under

the debugger, go to "Run>Start Debugging…," or type ⌘-Enter. Xamarin will start the server and open a browser window to *http://127.0.0.1:80080* (possibly with a different port), which at this point will just show the default "404 Not Found" page.

First steps

Alright, now that we have a project ready for some code, let's get our Nancy application up and running. Here's what it looks like to do perform a simple request using Nancy and Octokit.

```
using Nancy;
using Octokit;
using System;
using System.Collections.Generic;
using System.Linq;

namespace NancyApp
{
    public class Handler : NancyModule // ❶
    {
        private readonly GitHubClient client =
            new GitHubClient(new ProductHeaderValue("MyHello")); // ❷

        public Handler()
        {
            Get["/{user}", true] = async (parms, ct) => // ❸
                {
                    var user = await client.User.Get(parms.user.ToString()); // ❹
                    return String.Format("{0} people love {1}!",
                                         user.Followers, user.Name); // ❺
                };
        }
    }
}
```

❶ Here we derive a class from `NancyModule`, which is all you have to do to start receiving and processing HTTP requests in Nancy.

❷ The `GitHubClient` class is the entry point for Octokit. Here we create an instance which we'll use later on, using a placeholder product name – this name will not be used for the APIs we'll be accessing.

❸ The module's constructor needs to set up route mappings. We map `/{user}` to a lambda function using the `Get` dictionary that comes with `NancyModule`. The second parameter to the index operator says that the handler will be asynchronous.

❹ Here we see how to get the `{user}` part of the request URL (it comes as a property on the `parms` parameter), and how to query the GitHub User API using Octokit. Note that we have to `await` the result of the network query, since it may take some time.

❺ Nancy request handlers can simply return a text string, which will be marked as HTML for the viewing browser. Here we return a simple string with the user's name and number of followers.

The `async` and `await` keywords bear special mention. These comprise a syntactic nicety that encapsulates a series of functions that are running on an event loop. The code looks like it runs in order, but really when the `await` keyword is reached, the system starts an asynchronous request, and returns control back to the main event loop. Once the request has finished, and the promise is fulfilled, the event loop will then call back into the code that's expecting the return value of the `await` keyword, with all the scope variables intact. This feature was introduced in .NET 4.0 (which was released in 2012), and it lets you write asynchronous code almost as though it were synchronous. This is but one of the features that make C# the favorite of many developers.

This example is a bit more complicated than "hello, world," but it's still fairly succinct and clear. This bodes well, because we're about to introduce some complexity, in the form of OAuth.

OAuth flow

In order to post a status update for a commit, we're going to have to ask the user for permission. Apart from asking for their username and password (which gives way too much control, and if two-factor authentication is enabled may not even be enough), the only way to do this is OAuth, which isn't entirely straightforward.

Here's a simple outline of the OAuth process, from our little server's point of view:

1. We need an authorization token, either because we don't have one, or because the one we have is expired. This is just a string of characters, but we can't generate it ourselves, so we ask GitHub for one. This involves redirecting the user's browser to a GitHub API endpoint, with the kind of permission we're asking for and some other details as query parameters.
2. GitHub tells the user (through their browser) that an application is requesting some permissions, and they can either allow or deny them.
3. If the user allows this access, their browser is redirected to a URL we specified in step 1. A "code" is passed as a query parameter; this is not the access token we want, but a time-limited key to get one.

4. From inside the handler for this request, we can use a REST API to get the actual OAuth access token, which we can store somewhere safe. We do this because if we already have a token, we can skip all the way to the last step of this process.
5. Now we have permission, and we can use the GitHub API in authenticated mode.

This might seem overly complicated, but its design achieves several goals. First, permission can be scoped – an application is almost never given full access to the user's account and data. Second, the whole exchange is secure; at least one part of this has to go through the user, and cannot be automated. Third, the access token is never transmitted to the user's browser, which avoids an entire class of security vulnerabilities.

Let's walk through the code for our tiny little server's implementation of this flow. First, once we have a token, we should store it so we're not going through the entire redirect cycle for every user request. We're going to store it in a cookie (though since this goes back and forth to the user's browser, a production application would probably use a database). Nancy can help us with this, but first we have to enable it, and the way this is accomplished is by using a bootstrapper. We're going to add this class to our application:

```
using Nancy;
using Nancy.Bootstrapper;
using Nancy.Session;
using Nancy.TinyIoc;

namespace NancyApp
{
    public class Bootstrapper : DefaultNancyBootstrapper
    {
        protected override void ApplicationStartup(TinyIoCContainer container,
                                                   IPipelines pipelines)
        {
            CookieBasedSessions.Enable(pipelines);
        }
    }
}
```

Nancy will automatically detect a bootstrapper class, and use it to initialize our server. Now, from within a `NancyModule`, we can use the `Session` property to store and retrieve values that are transmitted as cookies.

Next, we have to include our application's ID and secret in some of the requests, so we embed them in the code by adding these fields to the `Handler` class. If you don't have an application, visit *https://github.com/settings/developers* to create one and use `http://localhost:8080/authorize` (depending in your environment, the port number might be slightly different) for the callback URL – we'll see why in a bit.

```
        private const string clientId = "<clientId>";
        private const string clientSecret = "<clientSecret>";
```

Obviously, you should use values from your own API application if you're following along.

After that, we'll need a helper method that kicks off the OAuth process:

```
private Response RedirectToOAuth()
{
    var csrf = Guid.NewGuid().ToString();
    Session["CSRF:State"] = csrf; // ❶
    Session["OrigUrl"] = this.Request.Path; // ❷

    var request = new OauthLoginRequest(clientId)
        {
            Scopes = { "repo:status" }, // ❸
            State = csrf,
        };
    var oauthLoginUrl = client.Oauth.GetGitHubLoginUrl(request);
    return Response.AsRedirect(oauthLoginUrl.ToString()); // ❹
}
```

❶ CSRF stands for "cross-site request forgery." This is a mechanism by which we can be sure the OAuth request process really did originate from our site. The GitHub OAuth API will pass this value back to us when the user authorizes access, so we store it in the cookie for later reference.

❷ Storing the original URL in the session cookie is a UX feature; once the OAuth process has completed, we want to send the user back to what they were trying to do in the first place.

❸ `repo:status` is the permission set we're asking for. Note that we're also including our CSRF token in this object; this is so GitHub can give it back to us later for verification.

❹ Here we use Octokit to generate the redirect URL, and send the user's browser there.

`RedirectToOAuth` is a method that can be called from any route handler in our module, if it's discovered that the token is missing or invalid. We'll see how it's called a bit later, but for now let's follow the rest of the OAuth process.

In our GitHub application settings, we specified an authorization URL. In this case, we've specified `http://localhost:8080/authorize`, and that's where GitHub will redirect the user's browser if they decide to grant our application the permissions it's asking for. Here's the handler for that endpoint, which has been inserted into the module constructor:

```
Get["/authorize", true] = async (parms, ct) => // ❶
    {
        var csrf = Session["CSRF:State"] as string;
        Session.Delete("CSRF:State");
```

```
        if (csrf != Request.Query["state"]) // ❷
        {
            return HttpStatusCode.Unauthorized;
        }

        var queryCode = Request.Query["code"].ToString();
        var tokenReq =  new OauthTokenRequest(clientId, // ❸
                                              clientSecret,
                                              queryCode);
        var token = await client.Oauth.CreateAccessToken(tokenReq);
        Session["accessToken"] = token.AccessToken; // ❹

        var origUrl = Session["OrigUrl"].ToString();
        Session.Delete("OrigUrl");
        return Response.AsRedirect(origUrl); // ❺
    };
```

❶ This is how you map paths to handler functions in Nancy. Any class that derives from `NancyModule` has an indexable object for every HTTP verb, and you can attach a synchronous or asynchronous handler to any one of them. There are also ways to include dynamic portions of URLs, which we'll see later on.

❷ Here we verify the CSRF token we generated before. If it doesn't match, something shady is happening, so we return a 401.

❸ This is the REST call that converts our OAuth code to an access token. In order to verify that this really is our application asking for the token, we pass in both the client ID and secret, as well as the code given to us by GitHub.

❹ This is where we store the resulting token in the session cookie. Again, this wouldn't be a good idea for a real application, but for our purposes it'll do.

❺ Here we redirect the user back to what they were originally trying to do, with as little disruption as possible.

Once all that is done, we've got our token and are able to continue on our merry way. All our handlers have to do to trigger an OAuth sequence is to call `RedirectToOAuth()` if it's necessary, and we'll automatically return the user to where they were when the process completes.

Status handler

Having gone through all that OAuth business, we should now have a token that grants us permission to create commit statuses, so let's see what it takes to do that. We're going to add this handler to our Nancy module constructor:

```
    Get["/{user}/{repo}/{sha}/{status}", true] = async (parms, ct) => // ❶
        {
            var accessToken = Session["accessToken"] as string;
            if (string.IsNullOrEmpty(accessToken))
```

```
            return RedirectToOAuth(); // ❷
        client.Credentials = new Credentials(accessToken);

        CommitState newState = Enum.Parse(typeof(CommitState), // ❸
                                          parms.status,
                                          true);
        try
        {
            var newStatus = new NewCommitStatus // ❹
            {
                State = newState,
                Context = "example-api-app",
                TargetUrl = new Uri(Request.Url.SiteBase),
            };
            await client.Repository.CommitStatus.Create(parms.user, // ❺
                                                        parms.repo,
                                                        parms.sha,
                                                        newStatus);
        }
        catch (NotFoundException) // ❻
        {
            return HttpStatusCode.NotFound;
        }

        var template = @"Done! Go to <a href=""https://" // ❼
        + @"api.github.com/repos/{0}/{1}/commits/{2}/status"
        + @""">this API endpiont</a>";
        return String.Format(template,
                             parms.user, parms.repo, parms.sha);
    };
```

❶ Note the request path for this handler: a GET request to `localhost:8080/user/repo/<sha>` will create a new status. This is easy to test with the browser, but also makes it easy for web crawlers to unknowingly trigger this API. For this example it's okay, but for a real application you'd probably want to require this to be a POST request.

❷ Here's where our OAuth helper comes in. We redirect through the OAuth flow if the session cookie doesn't have an authorization token. It's not shown here, but we'd also want to do this if we get an authorization exception from any of the Octokit APIs.

❸ Here we're trying to parse the last segment of the request URL into a member of the `CommitState` enumeration. Octokit tries to maintain type safety for all of its APIs, so we can't just use the raw string.

❹ The `NewCommitStatus` object encapsulates all the things you can set when creating a new status. Here we set the state we parsed earlier, a hopefully-unique context value that identifies our service, and a not-very-useful target URL (which should really go to an explanation of how the result was derived).

❺ This is the REST call to create the new status, which is asynchronous.

❻ There are a number of exceptions that could be thrown from the API, but the biggest one we want to handle is the `NotFoundException`, which has been translated from the HTTP 404 status. Here we translate it back to make for a nice(r) experience for the user.

❼ If we succeed, we render a snippet of HTML and return it from our handler. Nancy sets the response's content-type to `text/html` by default, so the user will get a nice clickable link.

That's it! If you've typed all this into a project of your own, you should be able to run it under the debugger, or host it in an ASP.NET server, and create commit statuses for your projects by opening URLs in your browser.

We noted this a bit earlier, but it bears repeating: this particular example responds to GET requests for ease of testing, but for a real service like this you'd probably want creation of statuses to use a POST request.

Summary

Even if you haven't written a lot of code during this chapter, you've learned a lot of concepts.

You've seen the commit status API, and you've seen how it's used by continuous integration software, but you know that it can be used for much more. You can read and write statuses, and you know how the API server coalesces many statuses into a single pass/fail value, and you also know how to write your own multi-status calculation if the default one doesn't meet your needs. You also know what's behind the green checkmarks and red X's you see in your pull requests.

You've learned how the OAuth web flow works, and why it's designed the way it is. OAuth is the key to many other capabilities of the GitHub API, and it's the right thing to do with regards to trust and permissions. This will allow you to write truly world-class GitHub-interfacing applications, whether running on the web or on a user's device.

You've gained a passing knowledge of C#, including its package system, at least one IDE, lambda functions, object initializers, and more. C# really is a nice language, and if you use it for a while, you'll probably miss some of its features if you write in anything else.

You've seen NuGet, the .NET package manager, and had a peek at the multitudes of packages in this ecosystem. The capability you have here is astounding; libraries exist for many common activities, and lots of uncommon ones too, so no matter what you need to do, you're likely to find a NuGet package to help you do it.

You've learned about Nancy, with which you can quickly build any HTTP service, from a REST API to an HTML-based interface, and all with a compact syntax and intuitive

object model. If you've never been exposed to the Sinatra view of the world, this probably makes you think about web servers a bit differently, and if you have, you'll have a new appreciation for how this model can be idiomatically implemented.

And you've had an introduction to Octokit, a type-safe implementation of a REST API, with built-in asynchrony and OAuth helpers. This toolkit really does make working with the GitHub API as simple and straightforward as using any .NET library, including the ability to explore it using Intellisense.

In the next chapter we will look at using Ruby to create and build Jekyll blogs.

CHAPTER 6
Ruby and Jekyll

The GitHub Jekyll repository (*https://github.com/jekyll/jekyll*) calls itself a "blog-aware, static site generator in Ruby." What does this mean? Jekyll is a set of technologies for building web sites. There are probably as many tools for generating web sites as there are websites, though, so why does Jekyll deserve notice over others? The Jekyll tool provides just enough to build a beautiful site, but then, the authors stopped and kept the core Jekyll tool free of many possible but risky improvements. Many other web site tools layer more and more complicated processes and require complicated backends for hosting. Jekyll is an antidote to this way of thinking. You can layer many other pieces onto Jekyll and build ever more complex sites if you want to, but you don't have to. Often, thinking simple leads to a much more elegant way to do things.

More concretely, Jekyll specifies a format that will take a set of files and compile them into HTML. Jekyll builds on top of two proven tools: Markdown, a markup language which is surprisingly readable and expressive, and Liquid Templates, a simple programming language which gives you just enough components to build modern web pages requiring conditionals and loops, but safe enough that you can run untrusted pages on public servers. With these two technologies and agreement on a layout structure, Jekyll can build very complicated web sites paradoxically without requiring a complicated structure of files and technologies.

Jekyll works natively with GitHub because a Jekyll blog is stored as a Git repository. When you push files into GitHub from a repository which GitHub recognizes as a Jekyll site, GitHub automatically rebuilds the site for you. Jekyll is an open source generator and defines a format for your source files, a format which other tools can easily understand and operate upon. This means you can build your own tools to interact with a Jekyll blog. Combining an open source tool like Jekyll with a well written API like the GitHub API makes for some powerful publishing tools.

Oddly enough, Jekyll takes us back to the advent of the web when all pages were static, and often created by hand. Nowadays most modern sites are dynamically generated

using a database backend. Jekyll provides us with a tool that recognizes we only need to be dynamic when new content is created and not when a new visitor requests the same page as the previous visitor. This is unfortunately how many sites work right now, resulting in the same page being regenerated over and over, and has led to overwrought solutions like page caching which has layered more lipstick on the pig. By acknowledging these facts, there is a massive reduction in complexity of Jekyll sites and with it, immense freedom. It may feel like you have traveled back in time, because when you look at a site built by Jekyll, you are looking at a set of static files, backed by nothing more complex than the same web server technologies available before the dot com bust of 2000.

The Harmless Brew that Spawned Jekyll

Though you won't see it in the documentation, the popularity of Jekyll is due in large part to the desire of bloggers, especially very technical ones, wanting a blogging tool option other than Wordpress. For many bloggers, Wordpress is a poor choice of blogging tools. Of course, there is always language snobbery in any programming community, but Wordpress is built on a language, PHP, which is widely reviled by developers from various language communities. Because PHP is more accessible language than other languages, the choice of PHP has fostered a large community of plugin developers. Unfortunately, this community is fractured, and the plugins which arise from it are at best poorly documented and poorly integrated, and at worst, poorly designed and buggy. While Wordpress does often have a plugin for anything you need, the problem it solves often creates more problems in the long term when scalability or database optimization or security, for example, become concerns.

Jekyll is interesting because of the components which are not present when compared against Wordpress. Jekyll does not require a database. Jekyll does not require you know how to write in HTML. While Wordpress ostensibly advertises itself as a tool which can be used without knowledge of these technologies, talk to those of us who have struggled with recovery of a mangled database after installation of a new Wordpress plugin, or resolving scalability issues on a large Wordpress site, or analyzing and fixing broken HTML produced by the Wordpress editor. We'll tell you that, when using Wordpress, you don't need to know about MySQL or HTML at all, because it is already too late by then to fix whatever problem you are facing.

Jekyll responds to these concerns in a really elegant way. Instead of authoring in HTML, you author in a simple and readable language called Markdown, which the Jekyll engine converts automatically to HTML for you. Instead of storing posts and other data inside a MySQL database, you use the filesystem to store posts and layouts and then regenerate the entire site when infrequent changes are made. And, to faciliate interaction, you use client side JavaScript plugins instead of pushing that interaction into your database. Jekyll has found a sweet spot with a simple technology set that makes your life easier

and makes beautiful blogs and simple web sites. And, the biggest benefit of doing things the "Jekyll" way is that all dependencies can sit within a repository, and this means your entire site can be hosted on GitHub.

(Less Than) Evil Scientist Parents

Like many of the open source technologies in heavy usage at GitHub, jekyll was originally developed by Tom Preson Warner, one of the co-founders of GitHub, and Nick Quaranto, of 37 Signals, though there are now thousands of contributors to the Jekyll codebase. Like many open source projects, the strength of the tool comes not from the brilliance of the original developers or the brilliance of the idea, but the way that those original developers cultivated community and involvement among the users of the tool.

Operating Jekyll Locally

To really use jekyll, you'll need the `jekyll` gem. As we explain in the [Link to Come], we could install a ruby gem using this command:

```
$ gem install jekyll
```

There are two issues with doing installation this way. The first is that any commands we run inside the command line are lost to us and the world (other than in our private shell history file). The second is that if we are going to publish any of our sites to GitHub, we will want to make sure we are matching the exact versions of Jekyll and its dependencies so that a site that works on our local laptop also works when published into GitHub. If you don't take care of this, you'll occasionally get an email like this from GitHub:

```
The page build failed with the following error:

page build failed

For information on troubleshooting Jekyll see
https://help.github.com/articles/using-jekyll-with-pages#troubleshooting
If you have any questions please contact GitHub Support.
```

The fix for these two issues is a simple one. You've probably seen other chapters using a `Gemfile` to install ruby libraries. Instead of using a manual command like `bundle` to install from the command line, let's put this dependency into the Gemfile. Then, anyone else using this repository can run the command `bundle install` and install the correct dependencies. And, instead of using the `jekyll` gem directly, use the `github-pages` gem which synchronizes your jekyll gem versions with those on GitHub. If you do get the email above, run the command `bundle update` to make sure that everything is properly setup and synchronized and generally this will reproduce the issues on your local setup, which is a much faster place to fix them.

```
$ printf "gem 'github-pages' >> Gemfile
$ bundle install
```

Creating and managing your dependencies inside a Gemfile is the smart way to get your jekyll tool synced with the version running on GitHub.

Now we are ready to create a Jekyll blog.

A Jekyll Blog in 15 Minutes

Now that we have our required tools, let's create a simple blog. Run these commands.

```
$ jekyll new myblog
$ cd myblog
```

The `jekyll new` commands creates the necessary structure for a minimal jekyll blog. Taking a look inside the directory, you'll see a few files which comprise the structure of a basic Jekyll blog.

The `jekyll new` command installs two CSS files: one for the blog (`main.css`) and one for syntax highlighting (`syntax.css`). Remember, you are in full control of this site; the `main.css` file is simply boilerplate which you can completely throw away if it does not suit your needs. The syntax file helps when including code snippets and contains syntax highlighting CSS which prettifies many programming languages.

Installation of a new blog comes with a .gitignore file as well which contains one entry: `_site`. When you use the jekyll library to build your site locally, all files are by default built into the _site directory. This .gitignore file prevents those files from being included inside your repository as they are overwritten by the jekyll command on GitHub when your files are pushed up to GitHub.

The `jekyll new` command does not create or initialize a new git repository for you with your files. If you want to do this, you will need to use the `git init` command. The jekyll initialization command does create the proper structure for you to easily add all files to a git repository; just use `git add .; git commit` and your gitignore file will be added and configure your repository to ignore unnecessary files like the `_site` directory.

All your blog posts are stored in the `_posts` directory. Jekyll sites are not required to have a `_posts` directory (you can use jekyll with any kind of static site) but if you do include files in this directory jekyll handles them in a special way. If you look in the `_posts` directory now, you see that the jekyll initialization command has created your first post for you, something like `_posts/2014-03-03-welcome-to-jekyll.Markdown`. These posts have a special naming format: the title of the post (with any whitespace

replaced with hyphens) trailed by the date and then an extension (either `.Markdown` or `.md` for Markdown files, or `.textile` for Textile)

Your new jekyll blog also comes with a few HTML files: an `index.html` file which is the starting point for your blog, and several layout files which are used as wrappers when generating your content. If you look in the `_layouts` directory, notice there is a file named `default.html` and another named `post.html`. These files are the layout files, files which are wrapped around all generated content, like those from your Markdown formatted blog posts. For example, the `post.html` file is wrapped around the generated content of each file stored inside the `_posts` directory. First the markup content is turned into HTML and then the layout wrapper is applied. If you look inside each of the files inside the `_layouts` directory, you will see that each contains a placeholder with `{{ content }}`. This placeholder is replaced with the generated content from other files.

These placeholders are actually a markup language on their own: "Liquid Templating." Liquid Templating (or Liquid Markup) was developed and open sourced by Shopify, and is a safe way to include programmatic constructs (like loops and variables) into a template, without exposing the rendering context to a full fledged programming environment. Shopify wanted to build a way for untrusted users of their public facing systems to upload dynamic content but not worry that the markup language would permit malicious activity; for example, given a full fledged embedded programming language, they would open themselves to attack if a user wrote code to open network connections to sites on their internal networks. Templating languages like PHP or ERB (embedded ruby templates, popular with the Ruby on Rails framework) allow fully embedded code snippets and while this is very powerful when you have full control over your source documents, it can be dangerous to provide a mechanism where that embedded code could look like `system("rm -rf /")`. Liquid provides many of the benefits of embedded programming templates, without the dangers.

Lastly, your jekyll directory has a special file called `_config.yml`. This is the jekyll configuration file. Peering into it, you'll see it is very basic:

```
name: Your New Jekyll Site
markdown: redcarpet
pygments: true
```

We only have three lines to contend with and they are simple to understand: the name of our site, the Markdown parser used by our jekyll command, and whether to use pygments to do syntax highlighting.

To view this site locally run this command:

```
$ jekyll serve
```

This command builds the entirety of your jekyll directory, and then starts a mini web server to serve the files up to you. If you then visit `http://localhost:4000` in your web

browser, you will see something the front page of your site and a single blog post listed in the index.

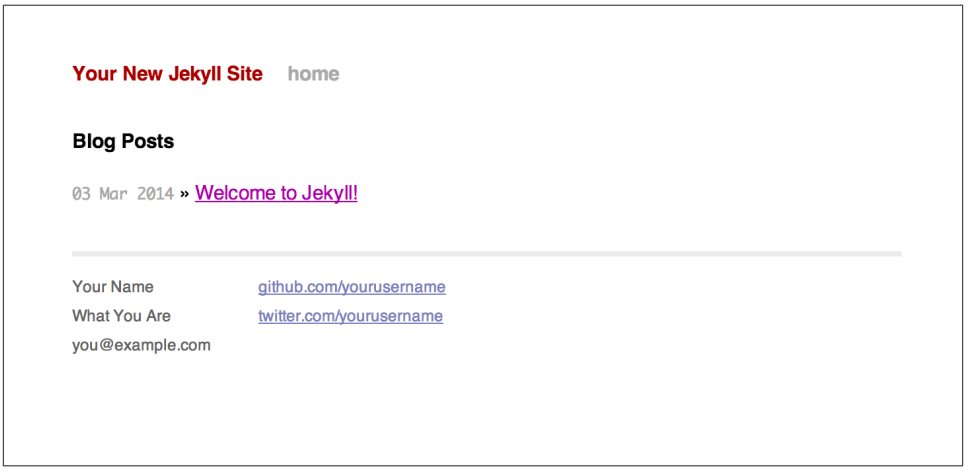


Figure 6-1. A bare Jekyll site

Clicking into the link inside the "Blog Posts" section, you will then see your first post.

Figure 6-2. A sample post

Our jekyll initialization command created this new post for us. This page is backed by the Markdown file inside the _posts directory which we saw earlier.

```
---
layout: post
title:  "Welcome to Jekyll!"
date:   2014-03-03 12:56:40
categories: jekyll update
---

You'll find this post in your `_posts` directory - edit this post and re-build (or run with the `-
To add new posts, simply add a file in the `_posts` directory that follows the convention: YYYY-MM

Jekyll also offers powerful support for code snippets:

{% highlight ruby %}
def print_hi(name)
  puts "Hi, #{name}"
end
print_hi('Tom')
#=> prints 'Hi, Tom' to STDOUT.
{% endhighlight %}

Check out the [Jekyll docs][jekyll] for more info on how to get the most out of Jekyll. File all b

[jekyll-gh]: https://github.com/mojombo/jekyll
[jekyll]:    http://jekyllrb.com
```

Hopefully you'll agree this is a fairly intuitive and readable alternative to raw HTML. This simplicity and readability is one of the major benefits of using Jekyll. Your source files maintain a readability that allows you to focus on the content itself, not on the technology that will eventually make them beautiful. Let's go over this file and investigate some of the important pieces.

YFM: YAML Front Matter

The first thing we see in a Jekyll file is the YAML Front Matter (YFM).

```
---
layout: post
title:  "Welcome to Jekyll!"
date:   2014-03-03 12:56:40
categories: jekyll update
---
```

YFM is a snippet of YAML ("YAML Aint Markup Language") delimited by three hyphens on both the top and bottom. YAML is a simple structured data serialization language used by many open source projects instead of XML. Many people find it more readable and editable by humans than XML. The YFM in this file shows a few configuration options: a layout, the title, the date and a list of categories.

The layout specified references one of the files in our `_layouts` directory. If you don't specify a layout file in the YFM, then Jekyll assumes you want to use a file called `de`

`fault.html` to wrap your content. You can easily imagine adding your own custom layout files to this directory and then overriding them in the YFM. If you look at this file, you see that it manually specifies the `post` layout.

The title is used to generate the `<title>` tag and can be used anywhere else you need it inside your template using the double braces syntax from Liquid: `{{ page.title }}`. Notice that any variable from the `_config.yml` file is prefixed with the `site.` namespace, while variables from your YFM are prefixed with `page.`. Though the title matches the filename (after replacing spaces with hyphens), changing the title in the YFM does not affect the name of the URL generated by Jekyll. If you want to change the URL, you need to rename the file itself. This is a nice benefit if you need to slightly modify the title and don't want to damage preexisting URLs.

The date and categories are two other variables included in the YFM. They are completely optional and strangely unused by the structure and templates created by default using the Jekyll initializer. They do provide additional context to the post, but are only stored in the Markdown file and not included inside the generated content itself. The categories list is often used to generate an index file of categories with a list of each post included in a category. If you come from a Wordpress background, you'll likely have used categories. These are generated dynamically from the MySQL database each time you request a list of them, but in Jekyll this file is staticly generated. If you wanted something more dynamic, you could imagine generating a JSON file with these categories and files, and then building a JavaScript widget which requests this file and then does something more interactive on the client side. Jekyll can take any template file and convert it to JSON (or any other format) — you are not limited to just generating HTML files.

YFM is completely optional. A post or page can be rendered into your Jekyll site without any YFM inside it. Without YFM, your page is rendered using the defaults for those variables, so make sure the default template, at the very least, is what you expect will wrap around all pages left with unspecified layouts.

One important default variable for YFM is the published variable. This variable is set to true by default. This means that if you create a file in your Jekyll repository and do not manually specify the published setting, it will be published automatically. If you set the variable to false then the post will not be published. With private repositories you can keep the contents of draft posts entirely private until writing has completed by making sure published is set to false. Unfortunately, not all tools that help you create Jekyll Markdown files remember to set the published variable explicitly inside of YFM, so make sure you check before committing the file to your repository if there is something you don't yet want published.

Jekyll markup

Going past the YFM, we can start to see the structure of Markdown files. Markdown files can be, at their simplest, just textual information without any formatting characters. In fact, if your layout files are well done, you can definitely create great blog posts without any fancing formatting, just pure textual content.

But, with a few small Markdown additions, you can really make posts shine. One of the first Markdown components we notice is the backtick character, which is used to wrap small spans of code (or code-ish information, like filenames in this case). As you use more and more Markdown, you'll find Markdown to be insidiously clever in the way it provides formatting characters without the onerous weight that HTML requires to offer the same explicit formatting.

```
Check out the [Jekyll docs][jekyll] for more info on how to get the most out of Jekyll. File all b

[jekyll-gh]: https://github.com/mojombo/jekyll
```

Links can be specified using `[format][link]`, where `link` is the fully qualified URL (like "http://example.com"), or a reference to a link at the bottom of the page. In our page we have two references, keyed as `jekyll-gh` and `jekyll`; we can then use these inside our page with syntax like `[Jekyll's GitHub repo][jekyll-gh]`. Using references has an additional benefit in that you can use the link more than once by its short name.

Though not offered in the sample, Markdown provides an easy way to generate headers of varying degrees. To add a header, use the `#` character, and repeat the `#` character to build smaller headers. These delimiters simply map to the H tag; two hash characters `##` turns into a `<h2>` tag. Building text enclosed by `<h3>` tags looks like `### Some Text`. You can optionally match the same number of hash symbols at the end of the line if you find it more expressive (`### Some Text ###`), but you don't have to.

Markdown offers easy shortcuts for most HTML elements: numbered and unordered lists, emphasis and more. And, if you cannot find a Markdown equivalent, you can embed normal HTML right next to Markdown formatting characters. The best way to write Markdown is to keep a Markdown cheat sheet near you when writing. John Gruber from Daring Fireball invented Markdown, and his site has a more in depth description of the how and why of Markdown.

Using the jekyll command

Running `jekyll --help` will show you the options for running jekyll. You already saw the `jekyll serve` command which builds the files into the `_site` directory and then starts a webserver with its root at that directory. If you start to use this mechanism to build your Jekyll sites then there are a few other switches you'll want to learn about.

If you are authoring and adjusting a page often, and switching back into your browser to see what it looks like, you'll find utility in the `-w` switch ("watch"). This can be used to automatically regenerate the entire site if you make changes to any of the source files. If you edit a post file and save it, that file will be regenerated automatically. Without the `-w` switch you would need to kill the jekyll server, and then restart it.

The jekyll watch switch does reload all HTML and markup files, but does not reload the _config.yml file. If you make changes to it, you will need to stop and restart the server.

If you are running multiple Jekyll sites on the same laptop, you'll quickly find that the second instance of `jekyll serve` fails because it cannot open port 4000. In this case, use `jekyll --port 4010` to open port 4010 (or whatever port you wish to use instead).

Privacy Levels with Jekyll

Jekyll repositories on GitHub can be either public or private repositories. If your repository is public you can host public content generated from the Jekyll source files without publishing the source files themselves. Remember, as noted previously, that any file without `publishing: false` inside the YFM will be made public the moment you push it into your repository.

Themes

Jekyll does not support theming internally, but it is trivial to add any CSS files or entire CSS frameworks. You could do this yourself, or you could just fork an existing jekyll blog which has the theming you like. The most popular themed Jekyll blog structure is Octopress. We don't display this here, but you another easy option is to add the Bootstrap CSS library just as we did in the [Link to Come] chapter.

Publishing on GitHub

Once you have your blog created, you can easily publish it to GitHub. There are two ways which you can publish Jekyll blogs:

- As a github.io site
- On a domain you own

Github offers free personal blogs which are hosted on the github.io domain. And, you can host any site with your own domain name with a little bit of configuration.

Using a GitHub.io Jekyll Blog

To create a github.io personal blog site, your Jekyll blog should be on the master branch of your Git repository. The repository should be named `username.github.io` on GitHub. If everything is setup correctly you can then publish your Jekyll blog by adding a remote for GitHub and pushing your files up. If you use the `hub` tool (a command for interacting with git and GitHub), you can go from start to finish with a few simple commands. Make sure to change the first line to reflect your username.

The hub tool was originally written in Ruby and as such could be easily installed using only `gem instal hub`, but hub was recently rewritten in Go. Go has a somewhat more complicated installation process, so we won't document it here. If you have the `brew` command installed for OSX, you can install hub with the `brew install hub` command. Other platforms vary, so check *http://github.com/github/hub* to determine the best way for your system.

Use thse commands to install your github.io hosted Jekyll blog.

```
$ export USERNAME=xrd
$ jekyll new $USERNAME.github.io
$ cd $USERNAME.github.io
$ git init
$ git commit -m "Initial checkin" -a
$ hub create  # You'll need to login here...
$ sleep $((10*60)) && open $USERNAME.github.io
```

The second to the last line creates a repository on GitHub for you with the same name as the directory. That last line sleeps for 10 minutes while your github.io site is provisioned on GitHub, and then opens the site in your browser for you. It can take ten minutes for GitHub to configure your site the first time, but subsequent content pushes will be reflected immediately.

Hosting On Your Own Domain

To host a blog on your own domain name, you need to use the `gh-pages` branch inside your repository. You need to create a CNAME file in your repository, and then finally establish DNS settings to point your domain to the GitHub servers.

The gh-pages branch

To work on the gh-pages branch, check it out and create the branch inside your repository.

```
$ git checkout -b gh-pages
$ rake post title="My next big blog post"
$ git add _posts
```

```
$ git commit -m "Added my next big blog post"
$ git push -u origin gh-pages
```

You will need to always remember to work on the gh-pages branch; if this repository is only used as a blog, then this probably is not an issue. Adding the `-u` switch will make sure that git always pushes up the gh-pages branch whenever you do a push.

The CNAME file

The CNAME file is a simple text file with the domain name inside of it.

```
$ echo 'mydomain.com' > CNAME
$ git add CNAME
$ git commit -m "Added CNAME"
$ git push
```

Once you have pushed the CNAME file to your repository, you can verify that GitHub thinks the blog is established correctly by visiting the admin page of your repository. An easy way to get there is using the `github` gem, no longer actively maintained but still a useful command line tool.

```
$ gem install github
$ github admin # Opens up https://github.com/username/repo/settings
```

The github gem is a useful command line tool, but unfortunately it is tied to an older version of the GitHub API, which means the documented functionality is often incorrect.

If your blog is correctly setup, you will see something like Figure 3 in the middle of your settings page.

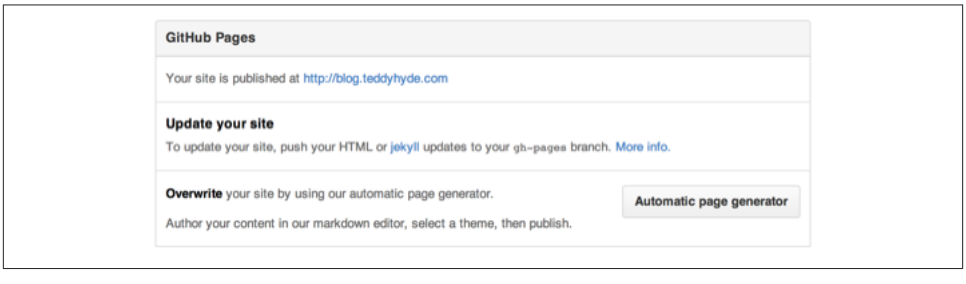


Figure 6-3. Settings for a Jekyll blog

GitHub has properly recognized the CNAME file and will accept requests made to that host on its servers. We are still not yet complete, however, in that we need to make sure the DNS is established for our site.

DNS Settings

Generally, establishing DNS settings for your site is straightforward. It is easiest if you are setting up DNS with a **subdomain** as opposed to an **apex domain**. To be more concrete, an apex domain is a site like mypersonaldomain.com, while a subdomain would be blog.mypersonaldomain.com.

Setting up a blog on a subdomain is simple: create a CNAME record in DNS that points to `username.github.io`.

For an apex domain, things are slightly more complicated. You must create DNS A records to point to these IP addresses: `192.30.252.153` and `192.30.252.154`. These are the IP addresses right now; there is always the possibility that GitHub could change these at some point in the future. For this reason, hosting on apex domains is risky. If GitHub needed to change their IP addresses (say during a denial of service attack), you would need to respond to this, and deal with the DNS propagation issues. If you instead use a subdomain, the CNAME record will automatically redirect to the correct IP even if that is changed by GitHub [1].

Importing from other blogs

There are many tools which can be used to import an existing blog into Jekyll. As Jekyll is really nothing more than a file layout convention, you just need to pull the relevant pieces (the post itself, and associated metadata like the post title, publishing date, etc.) and then write out a file with those contents. Jekyll blogs prefer Markdown, but they work fine with HTML content, so you can often convert a blog with minimal effort, and there are good tools which automate things for you.

From Wordpress

The most popular importer is the Wordpress importer. You will need the the *jekyll-import* gem, which is installed separately from the jekyll gem. If you have installed the `github-pages` gem then the importers are installed alongside the other tools packaged with this bundle.

Importing with direct database access

Once you have the `jekyll-import` gem, you can convert a Wordpress blog using a command like this:

```
$ ruby -rubygems -e 'require "jekyll-import";
    JekyllImport::Importers::WordPress.run({
      "dbname"   => "wordpress",
```

1. This is all well documented on the GitHub blog

```
      "user"     => "hastie",
      "password" => "lanyon",
      "host"     => "localhost",
      "status"         => ["publish"]
    })'
```

This command will import from an existing Wordpress installation, provided that your ruby code can access your database. This will work if you can log into the server itself and run the command on the server, or if the database is accessible across the network (which is generally bad practice when hosting Wordpress!).

Note the status option: this specifies that imported pages and posts are published automatically. More specifically, the YAML for each file will specify `published: true` which will publish the page or post into your blog. If you want to review each item individually, you can specify a status of `private` which will export the pages into Jekyll but leave them unpublished. Remember that if your repository is public, they will not be listed on the blog but can still be seen if someone peruses the source code for your blog on GitHub.

There are many more options than listed here. For example, by default, the Wordpress-Jekyll importer imports categories from your Wordpress database, but you can turn this off by specifying `"categories" => false`.

Importing from the Wordpress XML

Another alternative is to export the entire database as an XML file. Then, you can run the importer on that file.

```
ruby -rubygems -e 'require "jekyll-import";
    JekyllImport::Importers::WordpressDotCom.run({
      "source" => "wordpress.xml",
      "no_fetch_images" => false,
      "assets_folder" => "assets"
    })'
```

This can be used to export files from a server which you don't maintain, but works with sites you maintain and might be a more plausible option than running against a database.

To export the XML file, visit the export page on your site (*https://BLOGNAME.com/wp-admin/export.php*).

Like many free tools, there are definitely limitations to using this method of export. If your Wordpress site is anything beyond the simplest of Wordpress sites then using this tool to import from Wordpress means you will lose much of the metadata stored inside your blog. This metadata can include pages, tags, custom fields, and image attachments.

If you want to keep this metadata, then you might consider another import option like `Exitwp`. Exitwp is a python tool which provides a much higher level of fidelity between

the original Wordpress site and the final Jekyll site, but has a longer learning curve and option set.

Exporting from Wordpress alternatives

If you use another blog format other than Wordpress, chances are there is a Jekyll importer for it. Jekyll has dozens of importers, well documented on the Jekyll importer site *http://import.jekyllrb.com/*.

For example, this command line example from the importer site exports from Tumblr blogs.

```
$ ruby -rubygems -e 'require "jekyll-import";
    JekyllImport::Importers::Tumblr.run({
      "url"            => "http://myblog.tumblr.com",
      "format"         => "html", # ❶
      "grab_images"    => false,  # ❷
      "add_highlights" => false,  # ❸
      "rewrite_urls"   => false   # ❹
    })'
```

The Tumblr import plugin has a few interesting options.

❶ Write out HTML; if you prefer to use Markdown use `md`.

❷ This importer will grab images if you provide a true value.

❸ Wrap code blocks (indented 4 spaces) in a Liquid "highlight" tag if this is set to true.

❹ Write pages that redirect from the old Tumblr paths to the new Jekyll paths using this configuration option.

Exporting from Tumblr is considerably easier than Wordpress. The Tumblr exporter scrapes all public posts from the blog, and then converts to a Jekyll compatible post format.

We've seen how we can use the importers available on import.jekyllrb.com to import. What if we have a non-standard site that we need to import?

Scraping Sites into Jekyll

If you are stuck with a site that does not fit any of the standard importers, you could write your own importer by perusing the source of the Jekyll importers on GitHub. This is probably the right way to build an importer if you plan on letting others use it, as it will extend several jekyll importer classes already available to make importing standard for other contributors. Learning all the existing methods and reading through the dozens of samples can be a lot of work, however; another option is just to write out our files respecting the very simple format required by Jekyll. As we are programmers in the true

sense of the word we embrace and accept our laziness and choose the second route. Let's write some code to scrape and generate a Jekyll site.

Almost fifteen years ago while traveling in Brazil I grew increasingly frustrated with the guide books I used. It seemed like every time I went to a restaurant recommended by a guidebook I left the restaurant thinking "well, either they paid for that review or the review was written several years ago when this restaurant had a different owner." To address this discrepancy between reviews and realities, I built a site called ByTravelers.com. The idea was that travelers could use ByTravelers to record their experiences and easily share their experiences with their friends and families (replacing the long emails they used to send) and that that information would then become an authentication source of information about good and bad travel experiences.

I used an interesting programmable web server called Roxen featuring a dynamic language called Pike back when the web was dominated by one dynamic language: Perl, the "duct tape of the Internet." The site had ambitions greater than the talents of its architect, but suprisingly, a modest number of people found the site and started using it to track the experiences and simultaneously offer unbiased reviews of their traveling histories. It was exciting to see people creating content on their travels to China, Prague, and Cuba, and I vowed to visit Bhutan at least once in my lifetime after reading one particularly vivid account from an unknown contributor.

One of the problems with build on top of a web server and language like Roxen and Pike that never achieved critical mass in any way is that maintenance was a challenge. After moving back to the US and moving my hosting servers several times, I lost access to the source code (this was long before GitHub or even Git) and I lost access to the database and ByTravelers.com settled into oblivion.

Or, so I thought. One day I decided to look up ByTravelers on Archive.org, the Internet Archive. I found that almost all of the articles were listed there and available. Even though we have lost the source code and database, could we recover the site from just the Internet Archive? Let's scrape the articles from the Internet Archive and make them into a modern Jekyll site.

Jekyll Scraping Tactics

We'll use Ruby to scrape the site; Ruby has some intuitive gems like mechanize which provide automation of web clients. There is an API for the Internet Archive, but I found it flakey and unreliable. Scraping the site itself works well, but to reduce load on the archive, we'll cache our results using a gem called `VCR` (typically used to cached results from hitting a web service during test runs but perfectly capable here as well).

To write our parser, we will need to look at the structure of the archive presented on Archive.org. If we start on Archive.org, and enter "bytravelers.com" into the search box in the middle of the page, and then click "BROWSE HISTORY" we will be presented

with a calendar view which shows all the pages scraped by the Internet Archive for this site.

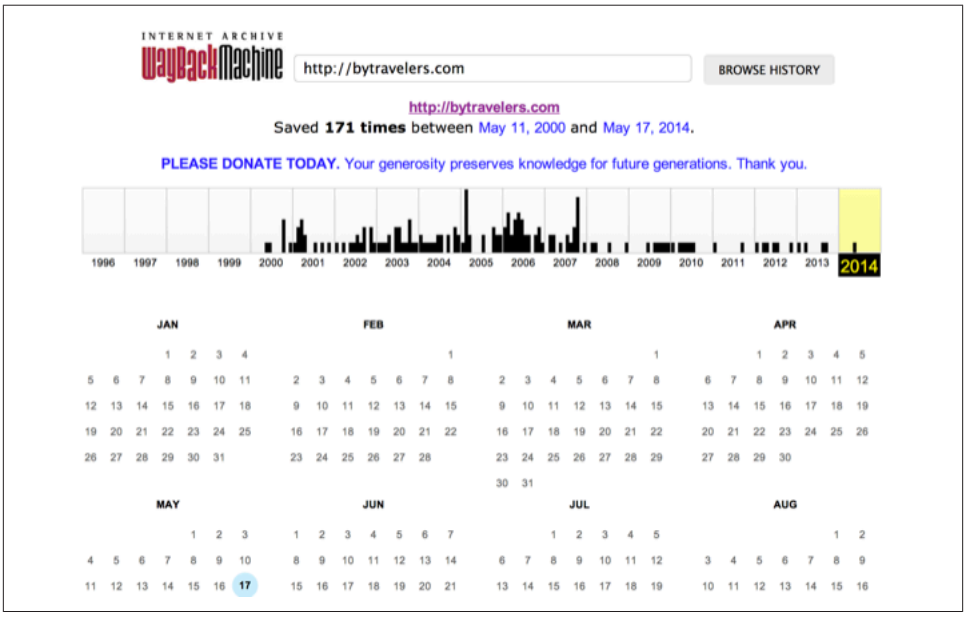


Figure 6-4. Calendar view of Archive.org

In the middle of 2003 I took down the server, intending to upgrade it to another set of technologies, and never got around to completing this migration, and then lost the data. If we click on the calendar item on June 6th, 2003, we will see a view of the data that was more or less complete at the height of the site's functionality and data. There are a few broken links to images, but otherwise the site is functionally archived inside Archive.org

Figure 6-5. Archive of ByTravelers.com on Archive.org

Taking the URL from Chrome, we can use this as our starting point for scraping. Clicking around throughout the site, it becomes evident that each URLs to a journal entry uses a standard format; in other words, `http://www.bytravelers.com/journal/entry/56` indicates the 56th journal item stored on the site. With this knowledge in hand, we can iterate over the first hundred or so URLs easily.

Going to one of these pages through the archived site, it is useful to view the source of the page and start to understand the structure of a page which we can then use when pointing our mechanize scraper at the page to pull out content. Any modern web browser supports a debug mode, and Chrome (my browser of choice) supports this as well. If we hold down the control key and click (at least on Mac OSX; righting-click on Windows or Linux works in the same way) into the "body" of a journal entry on its page, we will see a context menu that gives us the option to "Inspect Element". Chosing this option brings up the Chrome Developer Tools and shows us the HTML code of the page pretty printed for us. There are a few other items of note if we hover over any of the

printed items toward the bottom. As we moved our mouse over the items, we see a highlight applied to the page above, indicating the visual result once rendered in a browser of this specific this HTML code.

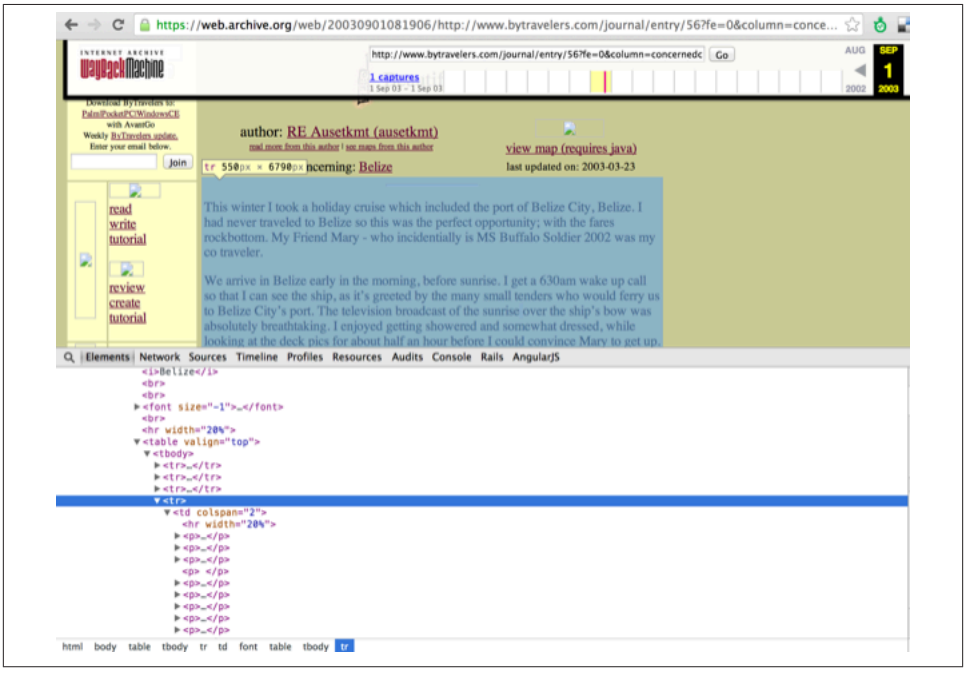


Figure 6-6. Inspecting Page Structure

Moving over different places in the HTML code displays different areas of our page; finding our way to the `tr` tag above the large number of `td` tags gives us access to the `body` of the post. Once there, you can see at the very bottom of the frame a hierarchy like `html body table tbody tr td font table tbody tr` which tells us clues about the path we need to take inside the DOM to reach this particular piece of content. With these indications in hand, we can start to write our parser to extract this text from pages scraped from the archive.

Writing our Parser

Let's start by writing a parser class.

```
require 'rubygems'
require 'mechanize'
require 'vcr'

VCR.configure do |c|  # ❶
  c.cassette_library_dir = 'cached'
```

```
  c.hook_into :webmock
end

class ByTravelersProcessor
  attr_accessor :mechanize  # ❷

  def initialize
    @mechanize = Mechanize.new { |agent| # ❸
      agent.user_agent_alias = 'Mac Safari'
    }
  end

  def run
    100.times do |i|
      get_ith_page( i ) # ❹
    end
  end

  def get_ith_page( i )
    puts "Loading #{i}th page"
  end

end
```

❶ VCR is a ruby library which caches HTTP requests. Typically used inside of tests, it is also an easy way to cache an HTTP request that you know will not change. Since these are pages stored inside an archive over ten years ago, it is safe to cache them, and the polite thing to do for an open system like Archive.org which relies on donations to pay for their bandwidth. The code you see here is boilerplate for configuring the VCR gem, loaded above.

❷ Our scraping is handled with the `mechanize` gem, and our class should maintain a reference to the scraper by declaring it here.

❸ After our class is instantiated, we hook into the class initialization stage and create our mechanize parser and assign it to the class.

❹ As we noted above, we have about 100 pages stored in the archive which we want to scrape. We loop 100 times over a function called `get_ith_page` which will do the scraping for us. Right now this function just prints out the index it is supposed to scrape.

Now that we have a harness for our scraper, let's install our scraper libraries.

Scraper Library Installation

Like other chapters which use Ruby, we create a `Gemfile` to manage our dependencies and then install them using the `bundle` command.

```
$ printf "source 'https://rubygems.org'\ngem 'vcr'\ngem 'mechanize'\ngem 'webmock'\n" >> Gemfile
$ bundle
```

With that we have the libraries we'll use to retrieve the content, so we can commence writing our wrapper script.

Parser Runner

Our runner is simple.

```
require 'rubygems'
require 'bundler/setup'
require './scraper'

btp = ByTravelersProcessor.new()
btp.run()
```

If we run this code now, we will just see our debug output.

```
$ ruby run.rb
...
Loading 91th page
Loading 92th page
Loading 93th page
Loading 94th page
Loading 95th page
Loading 96th page
Loading 97th page
Loading 98th page
Loading 99th page
...
```

Implementing Our Page Iterator

Now let's write the code which pulls out the information for the body and the title by implementing the `get_ith_page` method.

```
def get_ith_page( i )
  root = "https://web.archive.org/web/20030502080831/" +
    "http://www.bytravelers.com/journal/entry/#{i}"
  begin
    VCR.use_cassette("bt_#{i}") do # ❶
      @mechanize.get( root ) do |page|
        rows = ( page / "table[valign=top] tr" ) # ❷
        if rows and rows.length > 3
          self.process_body( i, rows[4] ) # ❸
        end
      end
    end
  rescue Exception => e
  end
end

def process_body( i, row )
  puts "#{i}: #{row.text().strip()[0...50]}" # ❹
end
```

❶ First, we load up a VCR cassette; this code says "store any HTTP requests inside my cassette directory (*cached*, specified in the configure page loaded by our script will be cached and saved in a file at the path `cached/bt_1.yml` (VCR adds the yml extension because it stores the results as a structured YAML file).

❷ Once we have loaded the page we see one of the powerful features of mechanize [2], a simple page searching syntax. If we provide the code `page / "table[valign=top] tr"` what we are doing is searching inside the page for a tag like `<table valign="top">` and then finding all the `<tr>` tags inside that. Recalling the DOM hierarchy we saw inside the Chrome Inspector, we can start to see how we will easily retrieve content from a scraped page.

❸ We then take the 2nd and 5th (ruby uses zero-based offsets) rows, and process it using a method called `process_body` which, you guessed it, processes the two rows as title and body respectively.

❹ The `process_body` method takes the retrieved DOM nodes and converts them to pure text (as opposed to leaving us with the HTML tags) and then strips whitespace from the front and end, and then prints the index in parentheses, the title and the first 50 characters of the body.

If we run our code now, we will see the body of our journal entries.

```
$ ruby run.rb
(4) Third day in Salvador :: I'm now entering the my third day in Salvador.  Th
(15) The Hill-Tribes of Northern Thailand :: I had heard about the hill-tribes in
northern Tha
(22) Passion Play of Oberammergau :: On Sunday, Sept. 17 Jeanne picked up Vic at Jackie
(23) "Angrezis in Bharat" :: Translation -  "Foreigners in India"Well since we
(24) Cuba - the good and bad :: April 1999Cuba, what an interesting place??!!
My a
(25) Nemaste :: Oct/Nov 2000"NEPAL"We spent our first 3 days in Ba
(26) Mexico/Belize/Guatemala :: Feb/Mar 1999Dear All
Well it´s been six weeks on t
(27) South Africa :: Apr/May 1999I got in from South Africa a few days
...
```

The first time we run this, we will see a slow march as Mechanize grabs each page from the Archive.org server. The next time we run our parser, however, things will be much faster. We have cached all the pages and so instead of doing a slow network crawl we are now speeding across our local filesystem, reading files from disk and simulating our network connection. Our client code does not have to change at all, the real power of the VCR gem.

2. Actually, this is handled by the Nokogiri parser, but Mechanizes exposes it transparently

Now let's break the title and body processing into separate methods. Add a method called `process_title` and add that into the `get_ith_page` method underneath the renamed `process` method now called `process_body`.

```
    end

    def write_post( page )
      title = page[0][0]
      image = page[0][1]
      body = page[1]
      creation_date = page[2]
      location = page[3]

      title.gsub!( /"/, '' )

      template = <<"TEMPLATE"
---
layout: post
title: "#{title}"
published: true
image: #{image}
location: #{location}
---

#{body}
TEMPLATE

      title_for_filename = title.downcase.gsub( /,+/, '' ).gsub( /[\s\/\:\;]+/, '-')
      # puts "Title: #{title_for_filename}"
      filename = "_posts/#{creation_date}-#{title_for_filename}.md"
      File.open( filename, "w+" ) do |f|
        f.write template
      end
    end

    def run
      100.times do |i|
        get_ith_page( i )
      end
      100.times do |i|
        if pages[i]
          write_post( pages[i] )
        end
      end
    end

    def process_creation_date( i, row )
      location, creation_date = row.text().split /last updated on:/
      creation_date.strip()
    end

    def process_location( i, row )
```

```
    location, creation_date = row.text().split /last updated on:/  # ❶

    location.gsub!( /Concerning: /, "" ) # ❷
```

We've modified the `get_ith_page` method to save each page as a tuple (the title and body) and then print them out after processing inside the `run` method.

❶ Our process body might look a little excessive. Why not just return the result of `row.text()`? The reason is that markdown is very specific about the format it requires for text formatting. Each block of text separated by two newlines will be formatted within `<p>` tags. Unfortunately, Mechanize and Nokogiri don't return text formatted that way, so this function retrieves each `<p>` tag we scraped, strips whitespace from the ends, and then adds it back to a body variable. If you scrape text from a site like we are doing here, you might need to normalize the text in a similar way.

❷ Do the same type of processing with the title. With this site there are occasionally titles which include the word "Title:" in the title itself (authors have to be forgiven for their own formatting quirks) so strip that out if we see it there.

❶ Keep track of each page in an array with each item of the array containing the title and body. We can then use this processed data later to build out our posts.

❷ At the end of processing, just print out the processed data we saved to verify we are finding the right structure in our pages.

If we re-run this script we will see identical output to the prior run, but now we are storing the information in an array and can use it to write out our Markdown files.

Generating Markdown

Now that we have our information parsed out, we can generate Markdown from it. As we've seen, Jekyll Markdown files are very simple: just a bit of YAML at the beginning, with text content following, formatted as Markdown.

We need to create a Git repository which we can push into GitHub. There is no reason why we cannot store our scraper scripts inside it, so let's just add the files to the same directory

```
$ git init
$ mkdir _posts
$ printf "_site" >> .gitignore
$ git add .gitignore
$ git commit -m "Initial checkin"
```

To generate Markdown posts, edit the `run` method to write out the files after we have retrieved and parsed the pages from Archive.org.

```
    def write_post( page )
      title = page[0]
```

```
    body = page[1]
    creation_date = page[2]

    title.gsub!( /"/, '' ) # ❶

    template = <<"TEMPLATE"  # ❷
---
layout: default
title: "#{title}"
published: false
---

#{body}
TEMPLATE

    title_for_filename = title.downcase.gsub( /,+/, '' ).gsub( /[\s\/\:\;]+/, '-') # ❸
    filename = "_posts/#{creation_date}-#{title_for_filename}.md"
    File.open( filename, "w+" ) do |f|
      f.write template
    end
  end

  def run
    100.times do |i|
      get_ith_page( i )
    end
    100.times do |i|
      if pages[i]
        write_post( pages[i] ) # ❹
      end
    end
  end

  def process_creation_date( i, row )
    location, creation_date = row.text().split /last updated on:/ # ❺
    creation_date.strip()
  end
```

❹ First, we modify the `run` method to call our new `write_post` method. This method is reponsible for writing out each datum in the array of processed data to a markdown file.

❶ We enclose the title inside the YAML inside of double quotes, so to make sure this does not conflict with the YAML parser we remove double quotes from the title here. We need to do this when we generate the filename later, so we use the `gsub!` method which modifies the string itself (rather than returning a new value but leaving the existing string intact).

❷ ❸ Inside the `write_post` method we create a "heredoc" template, and then stick values inside of it. Heredocs provide a more readable way to write out larger textual data, especially those with newlines. Heredocs simply start with a tag and then end the contents with the same tag ("TEMPLATE" here), and everything in between is treated as a single string.

❹ Jekyll expects markdown filenames to have a specific format. We need to modify the title by remove commas and quotes, and then converting whitespace, colons and semicolons to hyphen characters. We also need the creation date, which we retrieve in a later call, and parameterize the title with that information. Then we write out the file.

❺ Inside our new `process_creation_date` method we extract the creation date from the scraped data. We don't show it here because it is trivial, but after the `process_body` method call (which is inside the `get_ith_page` method) we added a call to this new function, giving it `i` and `row[3]` as the arguments.

We now have the posts generated properly, but we don't have an entry page into the blog. We can create a `index.md` file which just displays an index of all the blog posts. Inside this file we will use Liquid Tags to generate that list of posts. Notice that the site variable is populated with the list of posts automatically (Jekyll loads these up as long as they are in the `_posts` directory). We generate a link with the post URL, create a teaser from the content by generating a snippet of the body using the truncate method. Then we indicate the date that file was processed. Liquid provides a nice set of tools to convert and process text using the pipe character which allow you to build complex structures when combined with the looping constructs you see here.

There are a wide swath of constructs available with Liquid Templates. You can review the documentation for specifics. One common source of confusion when first learning Liquid is the difference between output tags and logic tags. Output tags use double braces surrounding the content (`{{ site.title }}`) while logic tags use a brace and percent symbol (`{% if site.title %}`). As you might expect, output tags place some type of visible output into the page, and logic tags perform some logic operation, like conditionals or loops.

```
---
layout: default
---

<h1>ByTravelers.com</h1>

Crowd sourced travel information.

<br/>

<div>
{% for post in site.posts %}
```

```
<a href="{{ post.url }}"><h2> {{ post.title }} </h2></a>
{{ post.content | strip_html | truncatewords: 40 }}
<br/>
<em>Posted on {{ post.date | date_to_string }}</em>
<br/>
{% endfor %}
</div>
```

The above template has both output and logic tags. We see a logic tag in the form of `{% for ... %}` which loops over each post. Jekyll will process the entire posts directory and provide it to pages inside the `site.posts` variable, and the `for` logic tag allows us to iterate over them. Remember that if we use a `{% for ... %}` tag we need to "close" the tag with a matching `{% endfor %}` tag. Inside of our for loop we have several output tags: `{{ post.url }}` outputs the post URL associated with a post, for example. We also have "filters" which are methods defined to process data. One such filter is the `strip_html` filter which you might guess strips out HTML text, converting it to escaped text. This is necessary when your text could include HTML tags. You'll also notice that filters can be "chained"; we process the body with the `strip_html` filter and then truncate the text by 40 characters using the `truncatewords:40` filter.

We also need to create a **default** layout, so create this inside the `_layouts` directory with the filename `default.html`.

```
<html>
<head>
<title>ByTravelers.com</title>
</head>

<body>

{{ content }}

</body>
</html>
```

This file is almost pure HTML, with only the `{{ content }}` tag. When we specify `default` as the layout inside YAML for a Markdown file, the Markdown text is converted to HTML, and then this layout file is wrapped around it. Notice this default layout is the same layout we have used inside our post files.

Finally, in order to convey that this is a Jekyll repository to both the command line Jekyll processor and GitHub service, we need to create a `_config.yml` file. We saw a simple version of this file earlier and can reuse this almost verbatim, changing only the name.

```
name: ByTravelers.com
markdown: redcarpet
pygments: true
```

Taking a moment to add our files to the Git repository, we can then take a look at our site using the `jekyll` command line tool.

```
$ git add .
$ git commit -m "Make this into a Jekyll site"
...
$ jekyll serve --watch
Configuration file: /Users/xrdawson/Projects/GithubBook/1234000000486/support/jekyll-parser/_confi
            Source: /Users/xrdawson/Projects/GithubBook/1234000000486/support/jekyll-parser
       Destination: /Users/xrdawson/Projects/GithubBook/1234000000486/support/jekyll-parser/_site
      Generating... done.
 Auto-regeneration: enabled
    Server address: http://0.0.0.0:4000
  Server running... press ctrl-c to stop.
```

We've started the Jekyll server in "watch" mode which means the site will be automatically regenerated if we edit the source files. Let's take a look at the site as currently configured on **http://localhost:4000**.

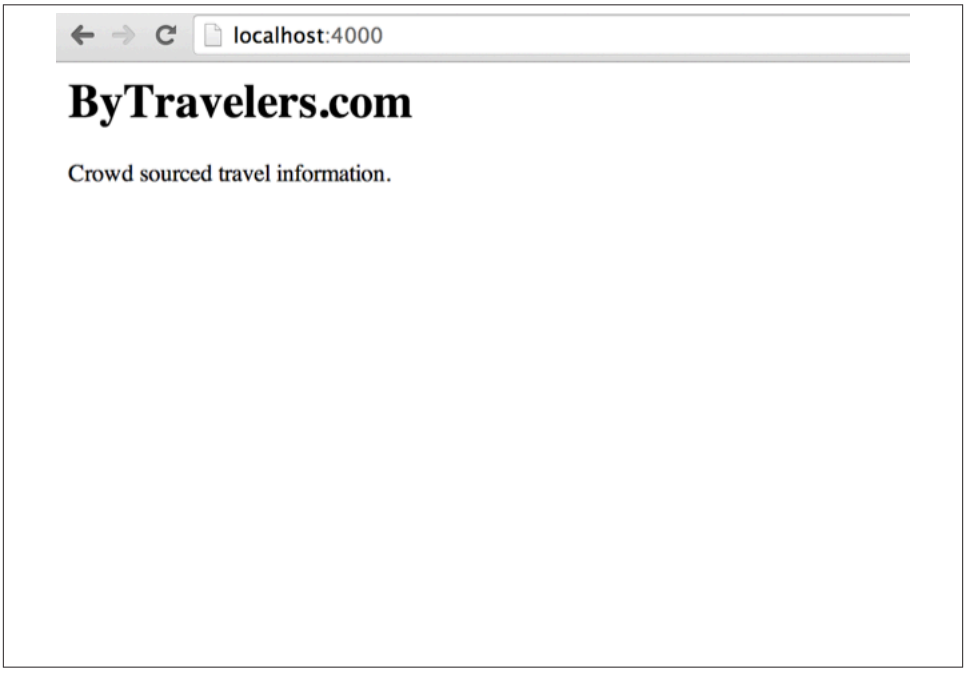


Figure 6-7. No posts at all

There are no posts! Ah, we elected to keep posts "unpublished" for now to review them before making them public. If we edit any file individually and change the line `published: false` inside our YFM to `published: true` then we will see that this file becomes

available inside our site. Let's do this for three of the files. Notice the server regenerates our files each time we change one of them.

```
...
      Regenerating: 1 files at 2014-06-20 12:54:52 ...done.
      Regenerating: 2 files at 2014-06-20 12:55:03 ...done.
      Regenerating: 1 files at 2014-06-20 12:55:15 ...done.
      Regenerating: 4 files at 2014-06-20 12:55:15 ...done.
...
```

And, if we reload our page we'll see them listed inside our index.

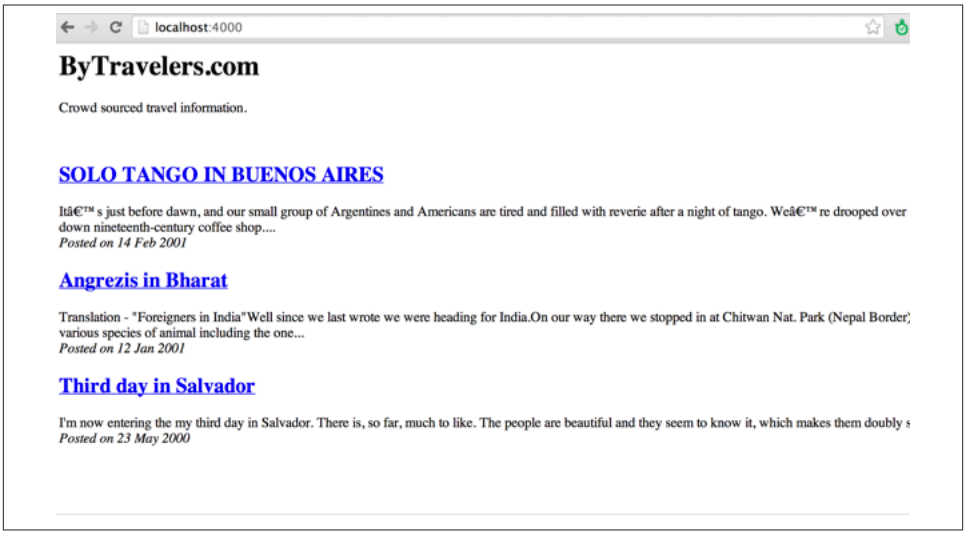


Figure 6-8. Publishing a single post

If we trust that all these posts are correct, we can change the `scraper.rb` script to make them all public (inside the heredoc template, just change the published flag), or we could change files individually by hand as we did here.

Taking a look at the blog post itself, we see this after clicking on the first link.

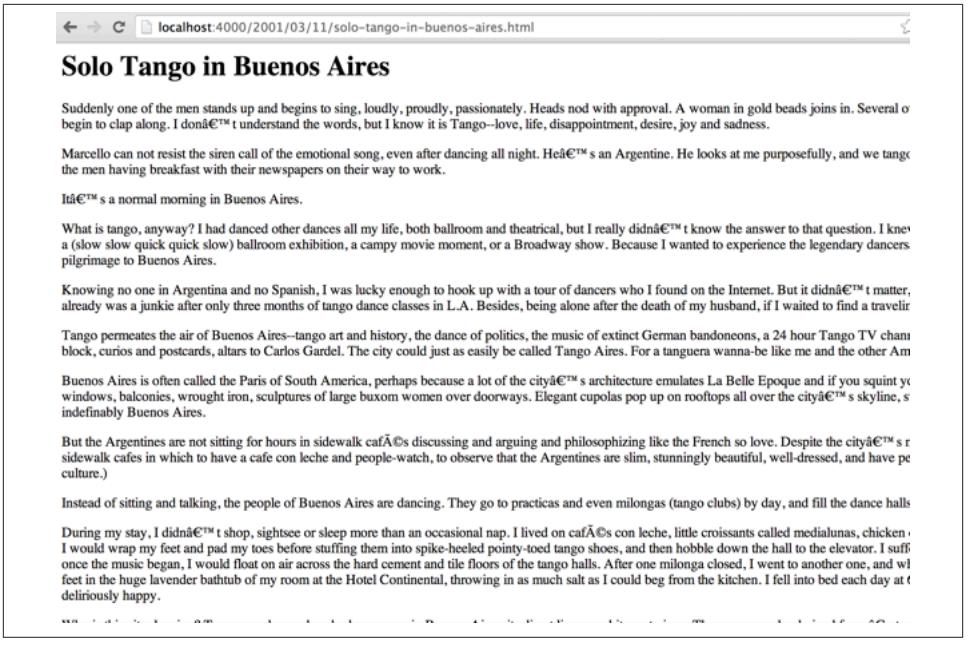


Figure 6-9. Not the best formatting

Not very pretty at all. We can beautiful this by adding some styling to the page. We'll use Bootstrap, the most popular CSS framework on GitHub. To start, edit the layout to include Bootstrap from the Bootstrap CDN. And, add a `container` class around the content.

```
<html>
<head>
<title>ByTravelers.com</title>

<link href="//maxcdn.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css" rel="stylesheet">

</head>

<body>

{{ content }}

</body>
</html>
```

Notice that our files are regenerated in the terminal window with `jekyll serve -w`. Refreshing the page shows some improvement, but we can do better. Let's make a front page which shows just the ten most recent post, and an archive page which shows all the posts in reverse chronological order.

First, copy the existing `index.md` file to a file named `archive.md`. Our current index looks just like our archive page needs to be. To make the front page with ten posts change the `{% for post in site.posts %}` tag to `{% for post in site.posts | limit:10 %}`. This limits us to ten posts. Add a link to `archive.html` at the bottom of our `index.md` file.

If we look at any of the pages we've scraped, they are now centered inside a box, but we don't have titles or anything else about the journal entry. Add a layout file just for posts by creating a file called `post.html` inside of the `_layouts` directory with the following contents.

```
---
layout: default
---

<h1>{{ page.title }}</h1>

{{ content }}
```

Notice also that this layout inherits the default layout. You can imagine wrapping many layouts within layouts to build up complicated output trees, but in this case we can now manage a base layout, including all the CSS and other complementary files associated with our site, and automatically propagate changes down into lower layout files.

Add these changes to the index, and commit them.

```
$ git add .
$ git commit -m "Added layout specific to posts"
```

Now that we have a post layout, we will need to adjust our post files to use this layout as right now they specify `layout: default` inside their YAML Front Matter. You might groan at the thought of editing all the files individually, but we don't need to go through that much effort as it turns out. If we make a one line change to our `scraper.rb` script (inside the `write_post` method) we can run the script again, and all our files will automatically be updated. As we committed them to our local git repository, we can also use tools like `git diff` to verify the changes we made were the correct ones.

```
    title.gsub!( /"/, '' )

    template = <<"TEMPLATE"
---
layout: post    #  <---- Set our layout variable to "post"
title: "#{title}"
published: true
---

#{body}
TEMPLATE
```

If we then run `ruby scraper.rb` we will see something like `Regenerating: 31 files at 2014-06-24 09:00:39 ...done.` indicating that our post files have changed. We can also verify that we made the correct changes by using the `git diff _posts` command. The result will be something like the following.

```
$ git diff _posts
diff --git a/support/jekyll-parser/_posts/2000-05-23-third-day-in-salvador.md b/support/jekyll-par
index 1873972..a4dbc21 100644
--- a/support/jekyll-parser/_posts/2000-05-23-third-day-in-salvador.md
+++ b/support/jekyll-parser/_posts/2000-05-23-third-day-in-salvador.md
@@ -1,10 +1,16 @@
 ---
-layout: post
+layout: post    #  <---- Set our layout variable to "post"
 published: true
 ---
...
```

The astute amongst you will also note that this shows we can add comments inside of our YAML Front Matter when needed.

Customizing Styling

Adding a CSS framework like Bootstrap helps things considerably, but we should match the original colors as well. The easiest way to get this information is to again view the site HTML using your browser's developer tools. On Chrome we can see that in the original body we hard code colors for the body, text and links.

Figure 6-10. Hard coding colors into our HTML

We need to convert this into the modern equivalent CSS and we can then add this a CSS file and include it inside all our pages by adding a link from the `default.html` template. Make a new directory inside called `assets/css` and write a file called `site.css` and include the CSS to match our original site. The choice of using `assets/css` as the directory structure is completely arbitrary; we could have chosen any structure we wanted and as long as the reference was correct it would have worked with our HTML templates.

```
body {
color: #000000;
background-color: #CCCC99;
}

a {
color: #603;
}
```

Then, modify the `default.html` template and include the new CSS file to allow our style changes to "cascade" to all our other files.

```
...
<title>ByTravelers.com</title>

<link href="//maxcdn.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css" rel="stylesheet">
<link href="/assets/css/site.css" rel="stylesheet">
```

```
</head>
...
```

Grabbing Original Images

Our site is bare beyond text and the original colors; adding the images would add some pop. We can easily modify our `scraper.rb` script and pull down the original images from our site and then republish them into our new Jekyll blog. Taking a look at the archived site, note that each title has an image to the left of it. If we customize our `process_title` method we can retrieve these images and then publish them into our blog.

Finding the image is easy: `img = ( title/ "img" )` will retrieve an "img" tag from the title as passed to us. Printing out the element using `puts` is simple way to view the contents of the elements, one of which looks like this `<img src="/web/20030502075943im_/http://www.bytravelers.com/images/pro/book.gif">` We can then dig into the element using syntax like `img.attr('src')` and get to the actual source of the image. We'll need to append the base site URL to this and can then retrieve the image from archive.org.

Unfortunately, the VCR gem does not easily allow us to make requests which are not captured. There are methods in VCR to ignore requests, but without heavily refactoring our `get_ith_method`. Instead we cheat by using a command line tool called `wget` to download the image. The VCR gem works by hooking into ruby libraries which make HTTP calls; by using the `wget` tool we can avoid using Ruby for a moment and download the file manually.

```
...
def process_title( i, title )
  img = ( title / "img" ) # ❶
  src = img.attr('src').text()
  filename = src.split( "/" ).pop

  output = "assets/images/"
  full = File.join( output, filename ) # ❷

  unless File.exists? full
    root = "https://web.archive.org"
    remote = root + src
    contents = `wget --quiet -O #{full} #{remote}`  # ❸
  end

  title = title.text()
  if title
    title.gsub!( /Title:/, "" )
    title.strip!
  end
  [ title, filename ]  # ❹
```

```
  end
  ...
```

❶ First, use the HTML node and extract the `img` tag from it. We then process the result, picking the `src` attribute from the tag, and then split it up by slash characters and pull off the last item returning just the filename.

❷ We then generate a path for the image. We chose `assets/images`, which keeps all our assets (like our CSS file) in a common place.

❸ The `wget` command requires a full remote URL and the path we created previously. We give it the `--quiet` switch to reduce noise during our processing.

❹ We will be placing the image into the template, so we want to return it with the title as processed data.

Once we have processed the information, we will need to modify the post template. We passed back the image inside the processed data from the `process_title` method. Ruby is an untyped language and this makes it so that even though initially we were passing back a string result from `process_title` and are now passing back an array of strings, we don't need to change the `get_ith_page` which assembles the results of our processing functions and puts them into the `page` array. When we iterate over the `page` result later, we should interpret the first element differently and pull the first item out as our title, and use the second item as the image for the post. This can all happen inside our `write_post` method.

```
...
  def write_post( page )
    title = page[0][0]  # ❶
    image = page[0][1]
    body = page[1]
    creation_date = page[2]

    title.gsub!( /"/, '' )

    template = <<"TEMPLATE"
---
layout: post
title: "#{title}"
published: true
image: #{image}   # ❷
---

#{body}
TEMPLATE

...
```

❶ As we mentioned, the first item in the array we receive is now the title and image packages as another array. We pull the first item out as the title text, and use the next item in the secondary array as the image.

❷ What do we do with the image? Let's put it into the YAML Front Matter. We can then utilize it from within our `post.html` layout file using Liquid tags.

Now, we modify our `post.html` layout to include the image near the title.

```
---
layout: default
---

<h1>{{ page.title }}</h1>

<img src="/assets/images/{{ page.image }}">

{{ content }}
```

We use the Liquid template tags `{{ page.image }}` to retrieve the image from our YFM and build an image tag inside our template.

We can also reuse this image on our index page (`index.md`).

```
---
layout: default
---

<h1>ByTravelers.com</h1>

Crowd sourced travel information.

<br/>

<div>
{% for post in site.posts %}
<a href="{{ post.url }}"><h2> {{ post.title }} </h2></a>
<img src="/assets/images/{{ post.image }}">
{{ post.content | strip_html | truncatewords: 40 }}
<br/>
<em>Posted on {{ post.date | date_to_string }}</em>
<br/>
{% endfor %}
</div>
```

Note that in this case the variable is presented as `post.image` as compared to `page.image` when we are inside a post. Jekyll and Liquid are not consistent here, so caveat emptor.

Our blog definitely has more life when we add in the original colors and images. It still looks like a blog from the last millenium, but it is an improvement.

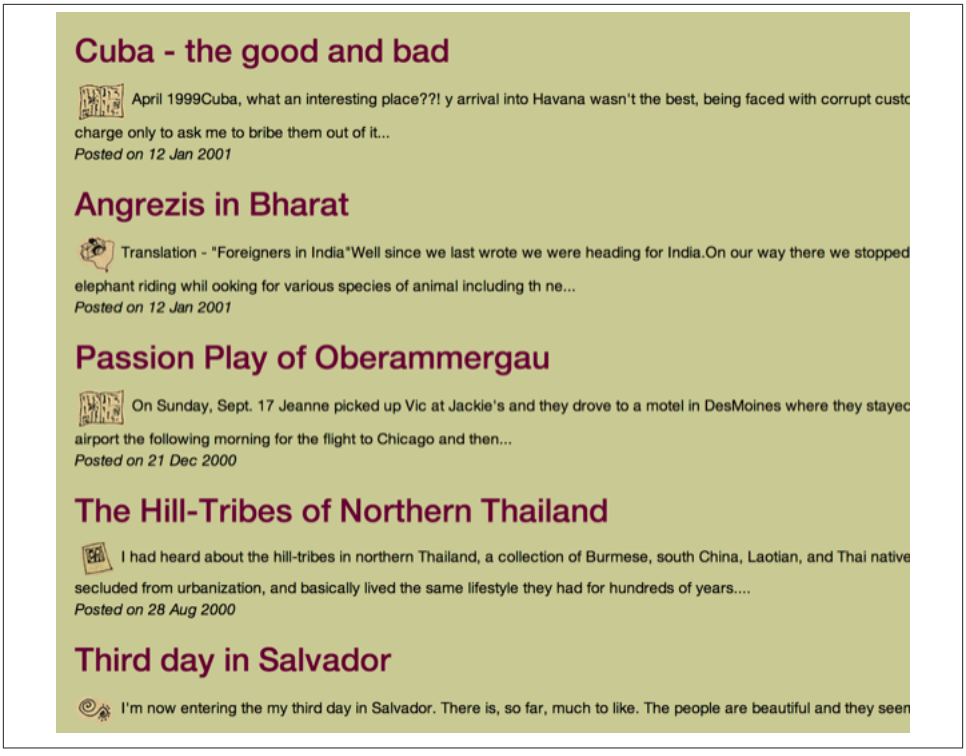


Figure 6-11. Restoring the original colors and images

Mapping Integration

This is a site about travel information, so it makes sense to add maps as well. We can process the location information we retrieved from the archived site and add a map image.

Add a new function called `process_location` and place it underneath the `process_creation_date` method inside the `get_ith_page` method.

```
...
def process_location( i, row )
  location, creation_date = row.text().split /last updated on:/  # ❶

  location.gsub!( /Concerning: /, "" ) # ❷
  location.strip!
  return location # ❸
end
...
```

❶ As we did in the `process_creation_date` method, we grab the row and split it in two pieces. This time we are only interested in the first element, the location.

❷ Each location string starts with the text "Concerning: " so we remove that here.

❸ Return the result

Now that we have the location, we need to insert it into the post itself. Once again we will place it inside the YFM. Modify the `write_post` method to grab the location from the array argument, and use that in the template inside the YFM.

```
...
  def write_post( page )
    title = page[0][0]
    image = page[0][1]
    body = page[1]
    creation_date = page[2]
    location = page[3]

    title.gsub!( /"/, '' )

    template = <<"TEMPLATE"
---
layout: post
title: "#{title}"
published: true
image: #{image}
location: #{location}
---

#{body}
TEMPLATE

...
```

Then, inside the `post.html` file, add an image tag with the map. We'll use a static image generated by Google Maps. We can utilize the location item inside the YFM and generate a map [3].

```
---
layout: default
---

<h1>{{ page.title }}</h1>


{% if page.location %}
<div>
<img src="http://maps.googleapis.com/maps/api/staticmap?center={{ page.location }}&zoom=13&size=40
</div>
{% endif %}
```

3. This idea was modified from a blog post on Katy Decorah's blog

```
<img src="/assets/images/{{ page.image }}">

{{ content }}
```

Inviting Contributions with GitHub "Fork"

When you publish a Jekyll blog, the fact that it is a repository on GitHub makes it simple to manage and track changes. In addition, because forking is a button click away, you can ask people to contribute or make changes with very little friction. You might have seen the banner saying "Fork me on GitHub" on many a software project page hosted on GitHub. We can motivate others to participate in our blog using pull requests. Let's add that as a final touch and invite people to make contributions the GitHub way. The GitHub blog first posted these banners and we'll use their code as-is inside our `default.html` page.

```
...
<body>

<a href="https://github.com/xrd/bytravelers.com"><img style="position: absolute; top: 0; right: 0;

<div class="container">
{{ content }}
...
```

Publishing our blog to GitHub

Like any other GitHub repository, we can then publish our blog using the same commands we saw with earlier repositories. Obviously you should change the username and blog name to suit your own needs.

```
$ export BLOG_NAME=xrd/bytravelers.com
$ gem install hub
$ hub create $BLOG_NAME # You might need to login here
$ sleep $((10*60)) && open $BLOG_NAME
```

And, don't forget to setup DNS records and give yourself appropriate time to let those records propagate out.

Summary

We've shown that we can quickly setup a blog on GitHub that has version control built in. We've shown how to import blogs like Wordpress into Jekyll. And, we've taken a site available only as an archive on the Internet Archive and scraped the content, images and even the colors, and then converted it to a Jekyll blog. Jekyll is a simple tool for managing websites, but a hidden benefit is that because Jekyll is so simple, you can easily write your own tools to interact and build on top of Jekyll. Once our site is a repository on GitHub, making changes yourself, or accepting them from other contributors is as easy as clicking "Merge" on a pull request.

In the next chapter we will continue looking at Jekyll by building an Android application that uses the Java GitHub API bindings and allows you to create Jekyll blog posts with the Git Data API.