The book you’re reading is still under development. As part of our Beta book program, we’re releasing this copy well before we ordinarily would. That way you’ll be able to get this content a couple of months before it’s available in finished form, and we’ll get feedback to make the book even better. The idea is that everyone wins!

Be warned. The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos. And there’s been no effort spent doing layout, so you’ll find bad page breaks, overlong lines, incorrect hyphenations, and all the other ugly things you wouldn’t expect to see in a finished book. We can’t be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you’ll enjoy it!

Throughout this process you’ll be able to download updated PDFs from your account on pragprog.com. When the book is finally ready, you’ll get the final version (and subsequent updates). In the meantime, we’d appreciate you sending us your feedback on this book at http://books.pragprog.com/titles/sdgrvr/errata.

Thank you for taking part in this experiment!

► Andy Hunt
Web services are everywhere these days. Once we as an industry figured out that XML travels over HTTP as well as HTML, we entered a new age of service-oriented architecture (SOA). This new way of grabbing data from remote sources means that developers must understand the mechanics of low-level TCP/IP and HTTP as well as the various higher-level XML dialects out in the wild: SOAP, REST, and XML-RPC. Luckily, Groovy helps us on all fronts.

In this chapter, we start with the low-level basics of how to determine our local TCP/IP address and domain name and those of remote systems. We move up the stack to HTTP—learning how to GET, POST, PUT, and DELETE programmatically. We end the chapter with examples of how to send and receive SOAP messages, XML-RPC messages, and RESTful requests. We’ll even parse a bit of comma-separated value (CSV) data just for old-times’ sake.

### 10.1 Finding Our Local IP Address and Name

```groovy
InetAddress.getLocalHost().getHostAddress
===> 63.246.7.76

InetAddress.getLocalHost().getHostName
===> myServer

InetAddress.getLocalHost().getCanonicalHostName
===> www.aboutgroovy.com
```

Before we can communicate with anyone else, it always helps knowing about ourselves. In this example, we discover our IP address, our local host name, and the DNS name that the rest of the world knows us as.
The InetAddress class comes to us from the java.net package. We cannot directly instantiate an InetAddress class (def addr = new InetAddress()) because the constructor is private. We can, however, use a couple of different static methods to return a well-formed InetAddress. The getLocalHost() method for getting local information is discussed here; getByName() and getAllByName() for getting remote information are discussed in Section 10.2, Finding a Remote IP Address and Domain Name, on the following page.

The getLocalHost() method returns an InetAddress that represents the localhost or the hardware on which it is running. As discussed in Section 5.2, Getter and Setter Shortcut Syntax, on page 72, getLocalHost() can be shortened to localhost in Groovy. Once we have a handle to localhost, we can call getHostAddress() to get our IP address or getHostName() to get the local machine name. This name is the private name of the system, as opposed to the name registered in DNS for the rest of the world to see. Calling getCanonicalHostName() performs a DNS lookup.

Of course, as discussed in Section 6.4, Running a Shell Command, on page 89, the usual command-line tools that ship with our operating system are just an execute() away. They might not be as easy to parse as the InetAddress methods, but as we can see they expose quite a bit more detail.

// available on all operating systems
"hostname".execute().text
==> myServer

// on Unix/Linux/Mac OS X
println "ifconfig".execute().text
==> en2: flags=8963<UP,BROADCAST,SMART,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500
inet6 fe80::21c:42ff:fe00:0%en2 prefixlen 64 scopeid 0x8
inet 10.37.129.3 netmask 0xffffff00 broadcast 10.37.129.255
ether 00:1c:42:00:00:00
media: autoselect status: active
supported media: autoselect

// on Windows
println "ipconfig /all".execute().text
==> Windows IP Configuration
Host Name .................: scottdavis1079
Primary Dns Suffix ........:
Node Type .................: Unknown
IP Routing Enabled........: No
WINS Proxy Enabled........: No
Finding a Remote IP Address and Domain Name

```
InetAddress.getByName("www.aboutgroovy.com")
==> www.aboutgroovy.com/63.246.7.76

InetAddress.getAllByName("www.google.com").each{println it}
==> www.google.com/64.233.167.99
www.google.com/64.233.167.104
www.google.com/64.233.167.147

InetAddress.getByName("www.google.com").hostAddress
==> 64.233.167.99

InetAddress.getByName("64.233.167.99").canonicalHostName
==> py-in-f99.google.com
```

In addition to returning information about the local machine, InetAddress can be used to find out about remote systems. `getByName()` returns a well-formed InetAddress object that represents the remote system. `getByName()` accepts either a domain name (for example, www.aboutgroovy.com) or an IP address (for example, 64.233.167.99). Once we have a handle to the system, we can ask for its `hostAddress` and its `canonicalHostName`.

Sometimes a DNS name can resolve to many different IP addresses. This is especially true for busy websites that load-balance the traffic among many physical servers. If a DNS name resolves to more than one IP address, `getByName()` will return the first one in the list, whereas `getAllByName()` will return all of them.

Of course, the usual command-line tools for asking about remote systems are available to us as well:
10.3 Making an HTTP GET Request

```groovy
def page = new URL("http://www.aboutgroovy.com").text
```
The simplest way to get the contents of an HTML page is to call `getText()` on the URL. This allows us to store the entire response in a String variable. If the page is too big to do this comfortably, we can also iterate through the response line by line using `eachLine()`.

Groovy adds a `toURL()` method to `java.lang.String`, allowing us to make identical requests using a slightly more streamlined syntax:

```
"http://www.aboutgroovy.com".toURL().text
"http://www.aboutgroovy.com".toURL().eachLine{...}
```

In Section 11.11, *Adding Methods to a Class Dynamically (ExpandoMeta-Class)*, on page 199, we’ll see how to streamline this to the point where we can simply call "http://www.aboutgroovy.com".get().

### Processing a Request Based on the HTTP Response Code

```groovy
def url = new URL("http://www.aboutgroovy.com")
def connection = url.openConnection()
if(connection.responseCode == 200){
    println connection.content.text
}
else{
    println "An error occurred:"
    println connection.responseCode
    println connection.responseMessage
}
```

Calling `getText()` directly on the URL object means that we expect everything to go perfectly—no connection timeouts, no 404s, and so on. Although we should be commend on our optimism, if we want to write slightly more fault-tolerant code, then we should call `openConnection()` on the URL. This returns a `java.net.URLConnection` object that will allow us to do a bit more detailed work with the URL. `connection.content.text` returns the same information as `url.text` while allowing us to do more introspection on the response—`connection.responseCode` for the 200 or the 404; `connection.responseMessage` for the OK or the File Not Found.

### Getting HTTP Response Metadata

```groovy
def url = new URL("http://www.aboutgroovy.com")
def connection = url.openConnection()
connection.responseCode
==> 200
connection.responseMessage
==> OK
connection.contentLength
```
=== 4216
connection.contentType
=== text/html
connection.date
=== 1191250061000
connection.expiration
=== 0
connection.lastModified
=== 0

connection.headerFields.each{println it}
===
Content-Length=[4216]
Set-Cookie=[JSESSIONID=3B2DE7CBDAE3D58EC46D5A8DF5AF89D1; Path=/]
Date=[Mon, 01 Oct 2007 14:47:41 GMT]
nul1=[HTTP/1.1 200 OK]
Server=[Apache-Coyote/1.1]
Content-Type=[text/html]

Once we have a handle to the URLConnection, we have full access to the accompanying response metadata. In addition to the responseCode and responseMessage, we can ask for things such as the contentLength and the contentType and can even iterate over each response header one by one.

Creating a Convenience GET Class

class Get{
    String url
    String queryString
    URLConnection connection
    String text

    String getText(){
        def thisUrl = new URL(this.toString())
        connection = thisUrl.openConnection()
        if(connection.responseCode == 200){
            return connection.content.text
        } else{
            return "Something bad happened\n" + "URL: " + this.toString() + "\n" + connection.responseCode + ": " + connection.responseMessage
        }
    }

    String toString(){
        return url + "?" + queryString
    }
}
def get = new Get(url: "http://search.yahoo.com/search")
get.queryString = "p=groovy"
println get
====> http://search.yahoo.com/search?p=groovy

println get.text
====> <html><head>...

get.url = "http://www.yahoo.com/no.such.page"
println get.text
====>
Something bad happened
URL: http://www.yahoo.com/no.such.page?p=groovy
404: Not Found

Up to this point we've been writing some pretty procedural\(^1\) code. It certainly gets the job done, but it suffers just a wee bit in terms of lack of reusability. (Don't you dare suggest that “copy and paste” is a valid type of reuse. You're a good object-oriented programmer—how could you even think such a thing?) This custom `Get` class wraps everything we've learned up to this point into something that can be reused. It has a nice simple interface and hides enough of the `HttpConnection` complexity to make it worth our time.

Now, nothing can compare to the simplicity of "http://www.aboutgroovy.com".toURL().text. On the opposite end of the spectrum is the Jakarta Commons `Http-Client\(^2\)—a great library that is far more complete than anything I could put together on my own. The drawback, of course, is adding yet another dependency to your project. Our custom `Get` class splits the difference nicely. It is slightly more robust than "".toURL().text, and yet it is implemented in pure Groovy so we don't have to worry about JAR bloat in our classpath.

One more thing: the `Get` class adds support for a `QueryString`. This is a collection of name/value pairs that can be appended to the end of an URL to further customize it. See Section 10.4, Working with QueryStrings, on the following page for more information.

**RESTful GET Requests**

"http://search.yahooapis.com/WebSearchService/V1/webSearch?
appid=YahooDemo&query=groovy&results=10".toURL().text

---
//alternately, using our Get class
def get = new Get()
get.url = "http://search.yahooapis.com/WebSearchService/V1/webSearch"
get.queryString = "appid=YahooDemo&query=groovy&results=10"
def results = get.text

There is a type of web service called RESTful web services. REST stands for Representational State Transfer. Although there are many differing interpretations of what it means to be truly RESTful, it is generally accepted that an HTTP GET request that returns XML results (as opposed to HTML or some other data format) constitutes the simplest form of a RESTful web service.

Yahoo offers a RESTful API that returns query results in XML. This query returns the top ten hits for the search term groovy. For the result of this query and how to parse it, see Section 10.12, Parsing Yahoo Search Results as XML, on page 177.

10.4 Working with QueryStrings

"http://search.yahoo.com/search?p=groovy".toURL().text

A QueryString allows you to make more complex HTTP GET requests by adding name/value pairs to the end of the address. Now instead of just asking for a static page at http://search.yahoo.com, we can make a dynamic query for all pages that contain the word groovy. The web is transformed from a simple distributed filesystem to a fully programmable web. The mechanics of programmatically making an HTTP GET request don’t change—it is no more complicated that what we were doing in Section 10.3, Making an HTTP GET Request, on page 156. However, the semantics of using QueryStrings opens up a whole new world of programmatic possibilities.

For example, complicated web pages like a Google map showing the Denver International Airport can be captured in a single URL. This means we can hyperlink it, bookmark it, or email it to a friend simply by clicking Link to This Page in the upper-right corner of the page. Each element in the QueryString represents a different aspect of the map: $l$ for the latitude/longitude center point of the map (39.87075, -104.39766),...
104.694214), z for the zoom level (11), t for the type (h, or hybrid), and so on.

"http://maps.google.com/maps?f=q&hl=en&geocode=&time=&date=&ttype=&q=dia&ssll=37.0625,-95.677068&sspn=34.038806,73.125&ie=UTF8&ll=39.87075,-104.694214&spn=0.2577,0.571289&z=11&iwloc=addr&om=1&t=h".toURL().text

### Building the QueryString from a List

```groovy
def queryString = []
queryString << "n=" + URLEncoder.encode("20")
queryString << "vd=" + URLEncoder.encode("m3")
queryString << "vl=" + URLEncoder.encode("lang_en")
queryString << "vf=" + URLEncoder.encode("pdf")
queryString << "p=" + URLEncoder.encode("groovy grails")
```

Often we'll be tasked with assembling a well-formed QueryString from an arbitrary collection of data values. The secret is to make sure the values are URL encoded (“foo bar baz” ==> foo+bar+baz), while the name portion (nonsense=) remains plain text. If we try to URLEncode the name and the value as a single string (“nonsense=foo bar baz”), the equals sign (=) will get converted to “%3D”, and your web server will most likely reject the request.

In this example, we create a `List` of name/value pairs, ensuring that only the value gets URLEncoded using the `java.net.URLEncoder`. Later when we need the well-formed QueryString, we call `queryString.join("&")`. As we saw in Section 4.14, `Join`, on page 60, this returns the list as a single string with each element joined by the string we passed in as the parameter.

This particular QueryString was built by performing an advanced Yahoo search and cherry-picking the interesting name/value pairs from the resulting URL. n returns 20 results instead of the default 10. vd limits the results to those posted in the past three months. vl returns only

---

Building the QueryString from a Map

def map = [n:20, vf:"pdf", p:"groovy grails"]
def list = []
map.each{name, value->
    list << "$name=" + URLEncoder.encode(value.toString())
}
println list.join("&")
==> n=20&vf=pdf&p=groovy+grails

Groovy Maps are a great way to represent QueryStrings since both naturally have name/value pairs. In this example, we still use a temporary List to store the URLencoded values and a join("&") to put them together at the last minute.

There is one edge case that keeps this from being a 100% solution. QueryStrings are allowed to have duplicate named elements, whereas Maps enforce unique names.


If you can live with this limitation, then Maps are the perfect solution. If you need to support duplicate named elements, see Section 10.4, Creating a Convenience QueryString Class for more information.

Creating a Convenience QueryString Class

class QueryString{
    Map params = [:]

    //this constructor allows you to pass in a Map
    QueryString(Map params){
        if(params){
            this.params.putAll(params)
        }
    }

    //this method allows you to add name/value pairs
    void add(String name, Object value){
        params.put(name, value)
    }

    //this method returns a well-formed QueryString
    String toString(){
        def list = []
        params.each{name, value->
            list << "$name=" + URLEncoder.encode(value.toString())
        }
        return list.join("&")
    }
}
10.14 Parsing an RSS Feed

```java
def rssFeed = "http://aboutgroovy.com/podcast/rss".toURL().text

Getting an RSS feed is as simple as making a plain old HTTP GET request.

//Response:
<rss xmlns:itunes="http://www.itunes.com/dtds/podcast-1.0.dtd" version="2.0">
  <channel>
    <title>About Groovy Podcasts</title>
    <link>http://aboutGroovy.com</link>
    <language>en-us</language>
    <copyright>2007 AboutGroovy.com</copyright>
    <itunes:subtitle>
      Your source for the very latest Groovy and Grails news
    </itunes:subtitle>
    <itunes:author>Scott Davis</itunes:author>
    <itunes:summary>About Groovy interviews</itunes:summary>
    <description>About Groovy interviews</description>
    <itunes:owner>
      <itunes:name>Scott Davis</itunes:name>
      <itunes:email>scott@aboutGroovy.com</itunes:email>
    </itunes:owner>
    <itunes:category text="Technology"/>
    <itunes:category text="Java"/>
    <itunes:category text="Groovy"/>
    <itunes:category text="Grails"/>
    <item>
      <title>AboutGroovy Interviews Neal Ford</title>
      <itunes:author>Scott Davis</itunes:author>
      <itunes:subtitle>Neal Ford of ThoughtWorks is truly a polyglot programmer. In this exclusive interview, Neal opines on Groovy, Ruby, Java, DSLs, and the future of programming languages. Opinionated and entertaining, Neal doesn't pull any punches. Enjoy.</itunes:subtitle>
      <itunes:summary url="http://aboutgroovy.com/podcasts/NealFord.mp3" length="33720522" type="audio/mpeg"/>
      <guid>http://aboutgroovy.com/podcasts/NealFord.mp3</guid>
      <pubDate>2007-04-17T01:15:00-07:00</pubDate>
      <itunes:duration>44:19</itunes:duration>
      <itunes:keywords>java,groovy,grails</itunes:keywords>
    </item>
  </channel>
</rss>

```
println it.pubDate
println it.enclosure.@url
println it.duration
println "-----"


AboutGroovy Interviews Neal Ford
2007-04-17T01:15:00-07:00
http://aboutgroovy.com/podcasts/NealFord.mp3
44:19
-----
AboutGroovy Interviews Jeremy Rayner
2007-03-13T01:18:00-07:00
http://aboutgroovy.com/podcasts/JeremyRayner.mp3
50:54
-----
...

XmlSlurper allows us to avoid dealing with the namespaces and extract the pertinent fields. See Section 8.9, Parsing an XML Document with Namespaces, on page 131 for more information.

Yahoo has a number of RSS feeds that offer more than simple blog syndication. See http://developer.yahoo.com/weather/ and http://developer.yahoo.com/traffic/ for a couple of examples of RSS feeds that send real data down the wire.
The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

**Groovy Recipes’s Home Page**
http://pragprog.com/titles/sdgrvr
Source code from this book, errata, and other resources. Come give us feedback, too!

**Register for Updates**
http://pragprog.com/updates
Be notified when updates and new books become available.

**Join the Community**
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

**New and Noteworthy**
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you’d like to have a paper copy of the book. It’s available for purchase at our store: pragprog.com/titles/sdgrvr.

Contact Us

Phone Orders: 1-800-699-PROG (+1 919 847 3884)
Online Orders: www.pragprog.com/catalog
Customer Service: orders@pragprog.com
Non-English Versions: translations@pragprog.com
Pragmatic Teaching: academic@pragprog.com
Author Proposals: proposals@pragprog.com