Sample code from Practicing Ruby Issue 7.2

To try it out, run the following command:

```
$ ruby http_server.rb
```

Then point your browser at http://localhost:2345.

```ruby
require 'socket'
require 'uri'

# Files will be served from this directory
WEB_ROOT = './public'

# Map extensions to their content type
CONTENT_TYPE_MAPPING = {
  'html' => 'text/html',
  'txt' => 'text/plain',
  'png' => 'image/png',
  'jpg' => 'image/jpeg'
}

# Treat as binary data if content type cannot be found
DEFAULT_CONTENT_TYPE = 'application/octet-stream'

# This helper function parses the extension of the
# requested file and then looks up its content type.

def content_type(path)
  ext = File.extname(path).split(".").last
  CONTENT_TYPE_MAPPING.fetch(ext, DEFAULT_CONTENT_TYPE)
end

# This helper function parses the Request-Line and
# generates a path to a file on the server.

def requested_file(request_line)
  request_uri  = request_line.split(" ")[1]
  path         = URI.unescape(URI(request_uri).path)

  clean = []

  # Split the path into components
  parts = path.split("/")
  parts.each do |part|
    # skip any empty or current directory (".") path components
    next if part.empty? || part == '.'
    # If the path component goes up one directory level (".."),
    # remove the last clean component.
    # Otherwise, add the component to the Array of clean components
    part == '..' ? clean.pop : clean << part
  end

  # return the web root joined to the clean path
  File.join(WEB_ROOT, *clean)
end

# Except where noted below, the general approach of
# handling requests and generating responses is
# similar to that of the "Hello World" example
# shown earlier.

server = TCPServer.new('localhost', 2345)

loop do
  socket       = server.accept
  request_line = socket.gets

  STDERR.puts request_line

  path = requested_file(request_line)
  path = File.join(path, 'index.html') if File.directory?(path)

  # Make sure the file exists and is not a directory
  # before attempting to open it.
  if File.exist?(path) && !File.directory?(path)
    File.open(path, "rb") do |file|
      socket.print "HTTP/1.1 200 OK\r\n" +
                   "Content-Type: #{content_type(file)}\r\n" +
                   "Content-Length: #{file.size}\r\n" +
                   "Connection: close\r\n"

      socket.print "\r\n"

      # write the contents of the file to the socket
      IO.copy_stream(file, socket)
    end
  else
```

```
    message = "File not found\n"

    # respond with a 404 error code to indicate the file does not exist
    socket.print "HTTP/1.1 404 Not Found\r\n" +
                 "Content-Type: text/plain\r\n" +
                 "Content-Length: #{message.size}\r\n" +
                 "Connection: close\r\n"

    socket.print "\r\n"

    socket.print message
  end

  socket.close
end
```

```
$> mkdir public
```

```
<html>
  <body>
    <div style="text-align: center">
      <h1>Hello World!</h1>
      <img src="earth_heart.jpg" />
    </div>
  </body>
</html>
```

put the earth_heart.jpg in there. See the photos section here in this group.

Article

This article was written by Luke Francl, a Ruby developer living in San Francisco. He is a developer at Swiftype where he works on everything from web crawling to answering support requests.

Implementing a simpler version of a technology that you use every day can help you understand it better. In this article, we will apply this technique by building a simple HTTP server in Ruby.

By the time you're done reading, you will know how to serve files from your computer to a web browser with no dependencies other than a few standard libraries that ship with Ruby. Although the server we build will not be robust or anywhere near feature complete, it will allow you to look under the hood of one of the most fundamental pieces of technology that we all use on a regular basis.

A (very) brief introduction to HTTP

We all use web applications daily and many of us build them for a living, but much of our work is done far above the HTTP level. We'll need come down from the clouds a bit in order to explore what happens at the protocol level when someone clicks a link to *http://example.com/file.txt* in their web browser.

The following steps roughly cover the typical HTTP request/response lifecycle:

1. The browser issues an HTTP request by opening a TCP socket connection to example.com on port 80. The server accepts the connection, opening a socket for bi-directional communication.

2. When the connection has been made, the HTTP client sends a HTTP request:

```
GET /file.txt HTTP/1.1
User-Agent: ExampleBrowser/1.0
Host: example.com
Accept: */*
```

3. The server then parses the request. The first line is the Request-Line which contains the HTTP method (GET), Request-URI (/file.txt), and HTTP version (1.1). Subsequent lines are headers, which consists of key-value pairs delimited by :. After the headers is a blank line followed by an optional message body (not shown in this example).

4. Using the same connection, the server responds with the contents of the file:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 13
Connection: close

hello world
```

5. After finishing the response, the server closes the socket to terminate the connection.

The basic workflow shown above is one of HTTP's most simple use cases, but it is also one of the most common interactions handled by web servers. Let's jump right into implementing it!

Writing the "Hello World" HTTP server

To begin, let's build the simplest thing that could possibly work: a web server that always responds "Hello World" with HTTP 200 to any request. The following code mostly follows the process outlined in the previous section, but is commented line-by-line to help you understand its implementation details:

```ruby
require 'socket' # Provides TCPServer and TCPSocket classes

# Initialize a TCPServer object that will listen
# on localhost:2345 for incoming connections.
server = TCPServer.new('localhost', 2345)

# loop infinitely, processing one incoming
# connection at a time.
loop do

  # Wait until a client connects, then return a TCPSocket
  # that can be used in a similar fashion to other Ruby
  # I/O objects. (In fact, TCPSocket is a subclass of IO.)
  socket = server.accept

  # Read the first line of the request (the Request-Line)
  request = socket.gets

  # Log the request to the console for debugging
  STDERR.puts request

  response = "Hello World!\n"

  # We need to include the Content-Type and Content-Length headers
  # to let the client know the size and type of data
  # contained in the response. Note that HTTP is whitespace
  # sensitive, and expects each header line to end with CRLF (i.e. "\r\n")
  socket.print "HTTP/1.1 200 OK\r\n" +
               "Content-Type: text/plain\r\n" +
               "Content-Length: #{response.bytesize}\r\n" +
               "Connection: close\r\n"

  # Print a blank line to separate the header from the response body,
  # as required by the protocol.
  socket.print "\r\n"

  # Print the actual response body, which is just "Hello World!\n"
  socket.print response

  # Close the socket, terminating the connection
  socket.close
end
```

To test your server, run this code and then try opening http://localhost:2345/anything in a browser. You should see the "Hello world!" message. Meanwhile, in the output for the HTTP server, you should see the request being logged:

```
GET /anything HTTP/1.1
```

Next, open another shell and test it with curl:

```
curl --verbose -XGET http://localhost:2345/anything
```

You'll see the detailed request and response headers:

```
* About to connect() to localhost port 2345 (#0)
*   Trying 127.0.0.1... connected
* Connected to localhost (127.0.0.1) port 2345 (#0)
> GET /anything HTTP/1.1
> User-Agent: curl/7.19.7 (universal-apple-darwin10.0) libcurl/7.19.7
              OpenSSL/0.9.8r zlib/1.2.3
> Host: localhost:2345
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 13
< Connection: close
<
Hello world!
* Closing connection #0
```

Congratulations, you've written a simple HTTP server! Now we'll build a more useful one.

Serving files over HTTP

We're about to build a more realistic program that is capable of serving files over HTTP, rather than simply responding to any request with "Hello World". In order to do that, we'll need to make a few changes to the way our server works.

For each incoming request, we'll parse the Request-URI header and translate it into a path to a file within the server's public folder. If we're able to find a match, we'll respond with its contents, using the file's size to determine the Content-Length, and its extension to determine the Content-Type. If no matching file can be found, we'll respond with a 404 Not Found error status.

Most of these changes are fairly straightforward to implement, but mapping the Request-URI to a path on the server's filesystem is a bit more complicated due to security issues. To simplify things a bit, let's assume for the moment that a requested_file function has been implemented for us already that can handle this task safely. Then we could build a rudimentary HTTP file server in the following way:

```
require 'socket'
require 'uri'

# Files will be served from this directory
WEB_ROOT = './public'

# Map extensions to their content type
CONTENT_TYPE_MAPPING = {
  'html' => 'text/html',
  'txt' => 'text/plain',
  'png' => 'image/png',
  'jpg' => 'image/jpeg'
}

# Treat as binary data if content type cannot be found
DEFAULT_CONTENT_TYPE = 'application/octet-stream'

# This helper function parses the extension of the
# requested file and then looks up its content type.

def content_type(path)
  ext = File.extname(path).split(".").last
  CONTENT_TYPE_MAPPING.fetch(ext, DEFAULT_CONTENT_TYPE)
end

# This helper function parses the Request-Line and
# generates a path to a file on the server.

def requested_file(request_line)
  # ... implementation details to be discussed later ...
end

# Except where noted below, the general approach of
# handling requests and generating responses is
# similar to that of the "Hello World" example
# shown earlier.

server = TCPServer.new('localhost', 2345)

loop do
  socket       = server.accept
  request_line = socket.gets

  STDERR.puts request_line

  path = requested_file(request_line)

  # Make sure the file exists and is not a directory
  # before attempting to open it.
  if File.exist?(path) && !File.directory?(path)
    File.open(path, "rb") do |file|
      socket.print "HTTP/1.1 200 OK\r\n" +
                   "Content-Type: #{content_type(file)}\r\n" +
                   "Content-Length: #{file.size}\r\n" +
                   "Connection: close\r\n"

      socket.print "\r\n"

      # write the contents of the file to the socket
      IO.copy_stream(file, socket)
    end
  else
    message = "File not found\n"

    # respond with a 404 error code to indicate the file does not exist
    socket.print "HTTP/1.1 404 Not Found\r\n" +
                 "Content-Type: text/plain\r\n" +
                 "Content-Length: #{message.size}\r\n" +
                 "Connection: close\r\n"

    socket.print "\r\n"

    socket.print message
  end

  socket.close
end
```

Although there is a lot more code here than what we saw in the "Hello World" example, most of it is routine file manipulation similar to the kind we'd encounter in everyday code. Now there is only one more feature left to implement before we can serve files over HTTP: the requested_file method.

Safely converting a URI into a file path

Practically speaking, mapping the Request-Line to a file on the server's filesystem is easy: you extract the Request-URI, scrub out any parameters and URI-encoding, and then finally turn that into a path to a file in the server's public folder:

```
# Takes a request line (e.g. "GET /path?foo=bar HTTP/1.1")
# and extracts the path from it, scrubbing out parameters
# and unescaping URI-encoding.
#
# This cleaned up path (e.g. "/path") is then converted into
# a relative path to a file in the server's public folder
# by joining it with the WEB_ROOT.
def requested_file(request_line)
  request_uri  = request_line.split(" ")[1]
  path         = URI.unescape(URI(request_uri).path)
  File.join(WEB_ROOT, path)
end
```

However, this implementation has a very bad security problem that has affected many, many web servers and CGI scripts over the years: the server will happily serve up any file, even if it's outside the WEB_ROOT.

Consider a request like this:

```
GET /../../../../etc/passwd HTTP/1.1
```

On my system, when File.join is called on this path, the ".." path components will cause it escape the WEB_ROOT directory and serve the /etc/passwd file. Yikes! We'll need to sanitize the path before use in order to prevent this kind of problem.

> **Note:** If you want to try to reproduce this issue on your own machine, you may need to use a low level tool like *curl* to demonstrate it. Some browsers change the path to remove the ".." before sending a request to the server.

Because security code is notoriously difficult to get right, we will borrow our implementation from Rack::File. The approach shown below was actually added to Rack::File in response to a similar security vulnerability that was disclosed in early 2013:

```
def requested_file(request_line)
  request_uri  = request_line.split(" ")[1]
  path         = URI.unescape(URI(request_uri).path)

  clean = []

  # Split the path into components
  parts = path.split("/")

  parts.each do |part|
    # skip any empty or current directory (".") path components
    next if part.empty? || part == '.'
    # If the path component goes up one directory level (".."),
    # remove the last clean component.
    # Otherwise, add the component to the Array of clean components
    part == '..' ? clean.pop : clean << part
  end

  # return the web root joined to the clean path
  File.join(WEB_ROOT, *clean)
end
```

To test this implementation (and finally see your file server in action), replace the requested_file stub in the example from the previous section with the implementation shown above, and then create an index.html file in a public/ folder that is contained within the same directory as your server script. Upon running the script, you should be able to visit http://localhost:2345/index.html but NOT be able to reach any files outside of the public/ folder.

Serving up index.html implicitly

If you visit http://localhost:2345 in your web browser, you'll see a 404 Not Found response, even though you've created an index.html file. Most real web servers will serve an index file when the client requests a directory. Let's implement that.

This change is more simple than it seems, and can be accomplished by adding a single line of code to our server script:

```
# ...
path = requested_file(request_line)

+ path = File.join(path, 'index.html') if File.directory?(path)

if File.exist?(path) && !File.directory?(path)
# ...
```

Doing so will cause any path that refers to a directory to have "/index.html" appended to the end of it. This way, / becomes /index.html, and /path/to/dir becomes path/to/dir/index.html.

Perhaps surprisingly, the validations in our response code do not need to be changed. Let's recall what they look like and then examine why that's the case:

```
if File.exist?(path) && !File.directory?(path)
  # serve up the file...
else
  # respond with a 404
end
```

Suppose a request is received for /somedir. That request will automatically be converted by our server into /somedir/index.html. If the index.html exists within /somedir, then it will be served up without any problems. However, if /somedir does not contain an index.html file, the File.exist? check will fail, causing the server to respond with a 404 error code. This is exactly what we want!

It may be tempting to think that this small change would make it possible to remove the File.directory? check, and in normal circumstances you might be able to safely do with it. However, because leaving it in prevents an error condition in the edge case where someone attempts to serve up a directory named index.html, we've decided to leave that validation as it is.

With this small improvement, our file server is now pretty much working as we'd expect it to. If you want to play with it some more, you can grab the complete source code from GitHub.

Where to go from here

In this article, we reviewed how HTTP works, then built a simple web server that can serve up files from a directory. We've also examined one of the most common security problems with web applications and fixed it. If you've made it this far, congratulations! That's a lot to learn in one day.

However, it's obvious that the server we've built is extremely limited. If you want to continue in your studies, here are a few recommendations for how to go about improving the server:

- According to the HTTP 1.1 specification, a server must minimally respond to GET and HEAD to be compliant. Implement the HEAD response.
- Add error handling that returns a 500 response to the client if something goes wrong with the request.
- Make the web root directory and port configurable.
- Add support for POST requests. You could implement CGI by executing a script when it matches the path, or implement the Rack spec to let the server serve Rack apps with `call`.
- Reimplement the request loop using GServer (Ruby's generic threaded server) to handle multiple connections.

Please do share your experiences and code if you decide to try any of these ideas, or if you come up with some improvement ideas of your own. Happy hacking!

We'd like to thank Eric Hodel, Magnus Holm, Piotr Szotkowski, and Mathias Lafeldt for reviewing this article and providing feedback before we published it.

> NOTE: If you'd like to learn more about this topic, consider doing the Practicing Ruby self-guided course on Streams, Files, and Sockets. You've already completed one of its reading exercises by working through this article!

> SEE ALSO: A similar HTTP server written in Python, contributed by Emily Horsman.