Twisted Network Programming Essentials, 2nd Edition

Originalartikel

Backup

<html> <meta charset=„UTF-8“/><meta http-equiv=„X-UA-Compatible“ content=„IE=edge“/><title>4. Web Servers - Twisted Network Programming Essentials, 2nd Edition [Book]</title><meta name=„viewport“ content=„width=device-width, initial-scale=1“/><meta property=„og:title“ content=„Twisted Network Programming Essentials, 2nd Edition“/><meta itemprop=„isPartOf“ content=„/library/view/twisted-network-programming/9781449326104/“/><meta itemprop=„name“ content=„4. Web Servers“/><meta property=„og:url“ itemprop=„url“ content=„https://www.oreilly.com/library/view/twisted-network-programming/9781449326104/ch04.html“/><meta property=„og:site_name“ content=„O’Reilly | Safari“/><meta property=„og:image“ itemprop=„thumbnailUrl“ content=„https://www.oreilly.com/library/cover/9781449326104/“/><meta property=„og:image:secure_url“ itemprop=„thumbnailUrl“ content=„https://www.safaribooksonline.com/library/cover/9781449326104/360h/“/><meta property=„og:description“ itemprop=„description“ name=„description“ content=„Chapter 4. Web Servers This chapter will first extend our experience with writing basic TCP servers to the construction of basic HTTP servers. With that context and understanding of the … - Selection from Twisted Network Programming Essentials, 2nd Edition [Book]“/><meta itemprop=„inLanguage“ content=„en“/><meta itemprop=„publisher“ content=„O'Reilly Media, Inc.“/><meta property=„og:type“ content=„article“/><meta property=„og:book:isbn“ itemprop=„isbn“ content=„9781449326111“/><meta property=„og:book:author“ itemprop=„author“ content=„Jessica McKellar“/><meta property=„og:book:author“ itemprop=„author“ content=„Abe Fettig“/><meta property=„og:book:tag“ itemprop=„about“ content=„Python“/><meta property=„og:image:width“ content=„400“/><meta property=„og:image:height“ content=„400“/><meta name=„twitter:card“ content=„summary“/><meta name=„twitter:site“ content=„@safari“/><header class=„global“ readability=„0“><svg xmlns=„http://www.w3.org/2000/svg“ viewbox=„0 0 59.13 9.88“><desc>O'Reilly logo</desc><rect x=„29.71“ y=„0.42“ width=„1.54“ height=„9.22“/></svg> </header><section id=„trial-overlay“> <h2 class=„trial-modal-title“>Stay ahead with the world's most comprehensive technology and business learning platform.</h2> <h2>With Safari, you learn the way you learn best. Get unlimited access to videos, live online training, learning paths, books, tutorials, and more.</h2> </section><section role=„document“ readability=„188“><section id=„sbo-reader“ readability=„29“><div class=„sbo-reader-content sbo-sample-reader“ readability=„126“> <section class=„chapter“ title=„Chapter 4. Web Servers“ epub:type=„chapter“ id=„twistedadn-CHP-4“ readability=„58“><h2 class=„title“>Chapter 4. Web Servers</h2> This chapter will first extend our experience with writing basic TCP servers to the construction of basic HTTP servers. With that context and understanding of the HTTP protocol in hand, we’ll then abandon the low-level API in favor of the high-level

```
twisted.web
```

APIs used for constructing sophisticated web servers. <h3 class=„title“>Note</h3> Twisted Web is the Twisted subproject focusing

on HTTP communication. It has robust HTTP 1.1 and HTTPS client and server implementations, proxy support, WSGI integration, basic HTML templating, and more. <div class=„sect1“ title=„Responding to HTTP Requests: A Low-Level Review“ readability=„83“> <h2 class=„title c1“ id=„twistedadn-CHP-4-SECT-1“>Responding to HTTP Requests: A Low-Level Review</h2> The HyperText Transfer Protocol (HTTP) is a request/response application-layer protocol, where requests are initiated by a client to a server, which responds with the requested resource. It is text-based and newline-delimited, and thus easy for humans to read. To experiment with the HTTP protocol we’ll create a subclass of

```
protocol.Protocol
```

, the same class we used to build our echo servers and clients in Chapter 2. Our protocol will know how to accept a connection, process the request, and send back an HTTP-formatted response. This section is intended as both a glimpse under the hood and a refresher on the HTTP protocol. When building real web servers, you’ll almost certainly use the higher-level

```
twisted.web
```

APIs Twisted provides. If you’d prefer to skip to that content, head over to Handling GET Requests. <div class=„sect2“ title=„The Structure of an HTTP Request“ readability=„83“> <h3 class=„title“ id=„twistedadn-CHP-4-SECT-1.1“>The Structure of an HTTP Request</h3> Every HTTP request starts with a single line containing the HTTP method, the path to the desired resource, and the HTTP version. Following this line are an arbitrary number of header lines. A blank line indicates the end of the headers. The header section is optionally followed by additional data called the body of the request, such as data being posted from an HTML form. Here’s an example of a minimal HTTP request. This request asks the server to perform the method

```
GET
```

on the root resource / using HTTP version 1.1: <pre class=„programlisting“>GET / HTTP/1.1 Host: www.example.com</pre> We can emulate a web browser and make this HTTP GET request manually using the telnet utility (taking care to remember the newline after the headers): <pre class=„programlisting“>$

```
telnet www.google.com 80
```

 Trying 74.125.131.99... Connected to www.l.google.com. Escape character is '^]'.

```
GET / HTTP/1.1 Host: www.google.com
```

</pre> The server responds with a line containing the HTTP version used for the response and an HTTP status code. Like the request, the response contains header lines followed by a blank line and the message body. A minimal HTTP response might look like this: <pre class=„programlisting“>HTTP/1.1 200 OK Content-Type: text/plain Content-Length: 17 Connection:

Close Hello HTTP world!</pre> www.google.com’s response is more complicated, since it is setting cookies and various security headers, but the format is the same. To write our own HTTP server, we can implement a

```
Protocol
```

that parses newline-delimited input, parses out the headers, and returns an HTTP-formatted response. Example 4-1 shows a simple HTTP implementation that echoes each request back to the client. Example 4-1. webecho.py <pre class=„kn“>from

```
twisted.protocols
```

```
import
```

```
basic
```

```
from
```

```
twisted.internet
```

```
import
```

```
protocol
```

```
,
```

```
reactor
```

```
class
```

```
HTTPEchoProtocol
```

```
(
```

```
basic
```

```
.
```

```
LineReceiver
```

```
):
```

```
  <code class="k">def</code> <code class="nf">__init__</code><code
class="p">(</code><code class="bp">self</code><code class="p">):</code>
      <code class="bp">self</code><code class="o">.</code><code
class="n">lines</code> <code class="o">=</code> <code class="p">[]</code>
```

```
  <code class="k">def</code> <code class="nf">lineReceived</code><code
class="p">(</code><code class="bp">self</code><code class="p">,</code> <code
class="n">line</code><code class="p">):</code>
      <code class="bp">self</code><code class="o">.</code><code
class="n">lines</code><code class="o">.</code><code
class="n">append</code><code class="p">(</code><code
class="n">line</code><code class="p">)</code>
      <code class="k">if</code> <code class="ow">not</code> <code
class="n">line</code><code class="p">:</code>
          <code class="bp">self</code><code class="o">.</code><code
class="n">sendResponse</code><code class="p">()</code>
  <code class="k">def</code> <code class="nf">sendResponse</code><code
class="p">(</code><code class="bp">self</code><code class="p">):</code>
      <code class="bp">self</code><code class="o">.</code><code
class="n">sendLine</code><code class="p">(</code><code class="s">"HTTP/1.1
200 OK"</code><code class="p">)</code>
      <code class="bp">self</code><code class="o">.</code><code
class="n">sendLine</code><code class="p">(</code><code
class="s">""</code><code class="p">)</code>
      <code class="n">responseBody</code> <code class="o">=</code> <code
class="s">"You said:</code><code class="se">\r\n\r\n</code><code
class="s">"</code> <code class="o">+</code> <code class="s">"</code><code
class="se">\r\n</code><code class="s">"</code><code class="o">.</code><code
class="n">join</code><code class="p">(</code><code
class="bp">self</code><code class="o">.</code><code
class="n">lines</code><code class="p">)</code>
      <code class="bp">self</code><code class="o">.</code><code
class="n">transport</code><code class="o">.</code><code
class="n">write</code><code class="p">(</code><code
class="n">responseBody</code><code class="p">)</code>
      <code class="bp">self</code><code class="o">.</code><code
class="n">transport</code><code class="o">.</code><code
class="n">loseConnection</code><code class="p">()</code>
```

```
class
```

```
HTTPEchoFactory
```

```
(
```

```
protocol
```

```
.
```

```
ServerFactory
```

```
):
```

```
  <code class="k">def</code> <code class="nf">buildProtocol</code><code
```

```
class="p">(</code><code class="bp">self</code><code class="p">,</code> <code
class="n">addr</code><code class="p">):</code>
      <code class="k">return</code> <code
class="n">HTTPEchoProtocol</code><code class="p">()</code>
```

```
reactor
```

```
.
```

```
listenTCP
```

```
(
```

```
8000
```

```
,
```

```
HTTPEchoFactory
```

```
())
```

```
reactor
```

```
.
```

```
run
```

```
()
```

</pre> <p>As with our basic TCP servers from Chapter 2, we create a protocol factory,

```
HTTPEchoFactory
```

, inheriting from

```
protocol.ServerFactory
```

. It builds instances of our

```
HTTPEchoProtocol
```

, which inherits from

```
basic.LineReceiver
```

so we don’t have to write our own boilerplate code for handling newline-delimited

protocols. We keep track of lines as they are received in

```
lineReceived
```

until we reach an empty line, the carriage return and line feed (

```
\r\n
```

) marking the end of the headers sent by the client. We then echo back the request text and terminate the connection. HTTP uses TCP as its transport-layer protocol, so we use

```
listenTCP
```

to register callbacks with the reactor to get notified when TCP packets containing our HTTP data arrive on our designated port. We can start this web server with python webecho.py then interact with the server through telnet or a web browser. Using telnet, the communication will look something like: <pre class=„programlisting“>$

```
telnet localhost 8000
```

 Trying 127.0.0.1… Connected to localhost. Escape character is '^]'.

```
GET / HTTP/1.1 Host: localhost:8000 X-Header: "My test header"
```

 HTTP/1.1 200 OK You said: GET / HTTP/1.1 Host: localhost:8000 X-Header: „My test header“ Connection closed by foreign host.</pre> It’s interesting to see what extra information your browser adds when making HTTP requests. To send a request to the server from a browser, visit http://localhost:8000. Figure 4-1 shows what I get when I make this request from Chrome on my MacBook. <div class=„figure-contents mediaobject“> Figure 4-1. Browser GET request By default, Chrome is telling websites about my operating system and browser and that I browse in English, as well as passing other headers specifying properties for the response. <div class=„sect2“ title=„Parsing HTTP Requests“ readability=„48“> <h3 class=„title“ id=„twistedadn-CHP-4-SECT-1.2“>Parsing HTTP Requests</h3> The

```
HTTPEchoProtocol
```

class in Example 4-1 understands the structure of an HTTP request, but it doesn’t know how to parse the request

and respond with the resource being requested. To do this, we’ll need to make our first foray into

```
twisted.web
```

. An HTTP request is represented by

```
twisted.web.http.Request
```

. We can specify how requests are processed by subclassing

```
http.Request
```

and overriding its

```
process
```

method. Example 4-2 subclasses

```
http.Request
```

to serve one of three resources: an HTML page for the root resource /, a page for /about, and a 404

```
http.NOT_FOUND
```

if any other path is specified. Example 4-2. requesthandler.py <pre class=„kn“>from

```
twisted.internet
```

```
import
```

```
reactor
```

```
from
```

```
twisted.web
```

```
import
```

```
http
```

```
class
```

```
MyRequestHandler
```

```
(
```

```
http
```

```
.
```

```
Request
```

```
):
```

```
  <code class="n">resources</code> <code class="o">=</code> <code
class="p">{</code>
      <code class="s">'/'</code><code class="p">:</code> <code
class="s">'&lt;h1&gt;Home&lt;/h1&gt;Home page'</code><code
class="p">,</code>
      <code class="s">'/about'</code><code class="p">:</code> <code
class="s">'&lt;h1&gt;About&lt;/h1&gt;All about me'</code><code
class="p">,</code>
      <code class="p">}</code>
  <code class="k">def</code> <code class="nf">process</code><code
class="p">(</code><code class="bp">self</code><code class="p">):</code>
      <code class="bp">self</code><code class="o">.</code><code
class="n">setHeader</code><code class="p">(</code><code class="s">'Content-
Type'</code><code class="p">,</code> <code class="s">'text/html'</code><code
class="p">)</code>
      <code class="k">if</code> <code class="bp">self</code><code
class="o">.</code><code class="n">resources</code><code
class="o">.</code><code class="n">has_key</code><code
class="p">(</code><code class="bp">self</code><code class="o">.</code><code
class="n">path</code><code class="p">):</code>
          <code class="bp">self</code><code class="o">.</code><code
class="n">write</code><code class="p">(</code><code
class="bp">self</code><code class="o">.</code><code
class="n">resources</code><code class="p">[</code><code
class="bp">self</code><code class="o">.</code><code
class="n">path</code><code class="p">])</code>
      <code class="k">else</code><code class="p">:</code>
          <code class="bp">self</code><code class="o">.</code><code
class="n">setResponseCode</code><code class="p">(</code><code
class="n">http</code><code class="o">.</code><code
class="n">NOT_FOUND</code><code class="p">)</code>
          <code class="bp">self</code><code class="o">.</code><code
class="n">write</code><code class="p">(</code><code class="s">"&lt;h1&gt;Not
Found&lt;/h1&gt;Sorry, no such resource."</code><code class="p">)</code>
      <code class="bp">self</code><code class="o">.</code><code
class="n">finish</code><code class="p">()</code>
```

```
class
```

```
MyHTTP
```

```
(
```

```
http
```

```
.
```

```
HTTPChannel
```

```
):
```

```
  <code class="n">requestFactory</code> <code class="o">=</code> <code
class="n">MyRequestHandler</code>
```

```
class
```

```
MyHTTPFactory
```

```
(
```

```
http
```

```
.
```

```
HTTPFactory
```

```
):
```

```
  <code class="k">def</code> <code class="nf">buildProtocol</code><code
class="p">(</code><code class="bp">self</code><code class="p">,</code> <code
class="n">addr</code><code class="p">):</code>
      <code class="k">return</code> <code class="n">MyHTTP</code><code
class="p">()</code>
```

```
reactor
```

```
.
```

```
listenTCP
```

```
(
```

```
8000
```

```
,
```

```
MyHTTPFactory
```

```
())
```

```
reactor
```

```
.
```

```
run
```

```
()
```

</pre> As always, we register a factory that generates instances of our protocol with the reactor. In this case, instead of subclassing

```
protocol.Protocol
```

directly, we are taking advantage of a higher-level API,

```
http.HTTPChannel
```

, which inherits from

```
basic.LineReceiver
```

and already understands the structure of an HTTP request and the numerous behaviors required by the HTTP RFCs. Our

```
MyHTTP
```

protocol specifies how to process requests by setting its

```
requestFactory
```

instance variable to

```
MyRequestHander
```

, which subclasses

```
http.Request
```

.

```
Request
```

’s

```
process
```

method is a noop that must be overridden in subclasses, which we do here. The HTTP response code is 200 unless overridden with

```
setResponseCode
```

, as we do to send a 404

```
http.NOT_FOUND
```

when an unknown resource is requested. To test this server, run python requesthandler.py; this will start up the web server on port 8000. You can then test accessing the supported resources, http://localhost:8000/ and http://localhost:8000/about, and an unsupported resource like http://localhost:8000/foo.   <div class=„sect1“ title=„Handling GET Requests“ readability=„57“> <h2 class=„title c1“ id=„twistedadn-CHP-4-SECT-2“>Handling GET Requests</h2> Now that we have a good grasp of the structure of the HTTP protocol and how the low-level APIs work, we can move up to the high-level APIs in

```
twisted.web.server
```

that facilitate the construction of more sophisticated web servers. <div class=„sect2“ title=„Serving Static Content“ readability=„43“> <h3 class=„title“ id=„twistedadn-CHP-4-SECT-2.2“>Serving Static Content</h3> A common task for a web server is to be able to serve static content out of some directory. Example 4-3 shows a basic implementation. Example 4-3. static_content.py <pre class=„kn“>from

```
twisted.internet
```

```
import
```

```
reactor
```

```
from
```

```
twisted.web.server
```

```
import
```

```
Site
```

```
from
```

```
twisted.web.static
import
File
resource
=
File
(
'/var/www/mysite'
)
factory
=
Site
(
resource
)
reactor
.
listenTCP
(
8000
,
factory
)
reactor
```

```
.
```

```
run
```

```
()
```

</pre> At this level we no longer have to worry about HTTP protocol details. Instead, we use a

```
Site
```

, which subclasses

```
http.HTTPFactory
```

and manages HTTP sessions and dispatching to resources for us. A

```
Site
```

is initialized with the resource to which it is managing access. A resource must provide the

```
IResource
```

interface, which describes how the resource gets rendered and how child resources in the resource hierarchy are added and accessed. In this case, we initialize our

```
Site
```

with a

```
File
```

resource representing a regular, non-interpreted file. <h3 class=„title“>Tip</h3>

```
twisted.web
```

contains implementations for many common resources. Besides

```
File
```

, available resources include a customizable

```
DirectoryListing
```

and

```
ErrorPage
```

, a

```
ProxyResource
```

that renders results retrieved from another server, and an

```
XMLRPC
```

implementation. The

```
Site
```

is registered with the reactor, which will then listen for requests on port 8000. After starting the web server with python static_content.py, we can visit http://localhost:8000 in a web browser. The server serves up a directory listing for all of the files in /var/www/mysite/ (replace that path with a valid path to a directory on your system). <div class=„sect3“ title=„Static URL dispatch“ readability=„20“> <h4 class=„title“ id=„id928285“>Static URL dispatch</h4> What if you’d like to serve different content at different URLs? We can create a hierarchy of resources to serve at different URLs by registering

```
Resource
```

s as children of the root resource using its

```
putChild
```

method. Example 4-4 demonstrates this static URL dispatch. Example 4-4. static_dispatch.py <pre class=„kn“>from

```
twisted.internet
```

```
import
```

```
reactor
```

```
from
```

```
twisted.web.server
```

```
import
```

```
Site
```

```
from
twisted.web.static
import
File
root
=
File
(
'/var/www/mysite'
)
root
.
putChild
(
"doc"
,
File
(
"/usr/share/doc"
))
root
.
putChild
(
"logs"
```

```
,
File
(
"/var/log/mysitelogs"
))
factory
=
Site
(
root
)
reactor
.
listenTCP
(
8000
,
factory
)
reactor
.
run
()
```

</pre> <p>Now, visiting http://localhost:8000/ in a web browser will serve content from /var/www/mysite, http://localhost:8000/doc will serve

content from /usr/share/doc, and http://localhost:8000/logs/ will serve content from /var/log/mysitelogs. These

```
Resource
```

hierarchies can be extended to arbitrary depths by registering child resources with existing resources in the hierarchy.   <div class=„sect2“ title=„Serving Dynamic Content“ readability=„44“> <h3 class=„title“ id=„twistedadn-CHP-4-SECT-2.3“>Serving Dynamic Content</h3> Serving dynamic content looks very similar to serving static content—the big difference is that instead of using an existing

```
Resource
```

, like

```
File
```

, you’ll subclass

```
Resource
```

to define the new dynamic resource you want a

```
Site
```

to serve. Example 4-5 implements a simple clock page that displays the local time when you visit any URL. Example 4-5. dynamic_content.py <pre class=„kn“>from

```
twisted.internet
```

```
import
```

```
reactor
```

```
from
```

```
twisted.web.resource
```

```
import
```

```
Resource
```

```
from
```

```
twisted.web.server
```

```
import
```

```
Site
```

```
import
```

```
time
```

```
class
```

```
ClockPage
```

```
(
```

```
Resource
```

```
):
```

```
  <code class="n">isLeaf</code> <code class="o">=</code> <code
class="bp">True</code>
  <code class="k">def</code> <code class="nf">render_GET</code><code
class="p">(</code><code class="bp">self</code><code class="p">,</code> <code
class="n">request</code><code class="p">):</code>
      <code class="k">return</code> <code class="s">"The local time is
</code><code class="si">%s</code><code class="s">"</code> <code
class="o">%</code> <code class="p">(</code><code class="n">time</code><code
class="o">.</code><code class="n">ctime</code><code class="p">(),)</code>
```

```
resource
```

```
=
```

```
ClockPage
```

```
()
```

```
factory
```

```
=
```

```
Site
```

```
(
```

```
resource
```

```
)
```

```
reactor
```

```
.
```

```
listenTCP
```

```
(
```

```
8000
```

```
,
```

```
factory
```

```
)
```

```
reactor
```

```
.
```

```
run
```

```
()
```

</pre>

```
ClockPage
```

is a subclass of

```
Resource
```

. We implement a

```
render_
```

method for every HTTP method we want to support; in this case we only care about supporting GET requests, so

```
render_GET
```

is all we implement. If we were to POST to this web server, we’d get a 405 Method Not Allowed unless we also implemented

```
render_POST
```

. The rendering method is passed the request made by the client. This is not an instance of

```
twisted.web.http.Request
```

, as in Example 4-2; it is instead an instance of

```
twisted.web.server.Request
```

, which subclasses

```
http.Request
```

and understands application-layer ideas like session management and rendering.

```
render_GET
```

returns whatever we want served as a response to a GET request. In this case, we return a string containing the local time. If we start our server with python dynamic_content.py, we can visit any URL on http://localhost:8000 with a web browser and see the local time displayed and updated as we reload. The

```
isLeaf
```

instance variable describes whether or not a resource will have children. Without more work on our part (as demonstrated in Example 4-6), only leaf resources get rendered; if we set

```
isLeaf
```

to

```
False
```

and restart the server, attempting to view any URL will produce a 404 No Such Resource. <div class=„sect2“ title=„Dynamic Dispatch“ readability=„68“> <h3 class=„title“ id=„twistedadn-CHP-4-SECT-3.1“>Dynamic Dispatch</h3> We know how to serve static and dynamic content. The next step is to be able to respond to requests dynamically, serving different resources based on the URL. Example 4-6 demonstrates a calendar server that displays the calendar for the year provided in the URL. For example, visiting http://localhost:8000/2013 will display the calendar for 2013, as shown in <a class=„xref“

href=„https://www.oreilly.com/library/view/twisted-network-programming/9781449326104/ch04.html#twistedadn-CHP-4-FIG-2“ title=„Figure 4-2. Calendar“>Figure 4-2. Example 4-6. dynamic_dispatch.py <pre class=„kn“>from

```
twisted.internet
```

```
import
```

```
reactor
```

```
from
```

```
twisted.web.resource
```

```
import
```

```
Resource
```

```
,
```

```
NoResource
```

```
from
```

```
twisted.web.server
```

```
import
```

```
Site
```

```
from
```

```
calendar
```

```
import
```

```
calendar
```

```
class
```

```
YearPage
```

```
(
```

```
Resource
```

```
):
```

```
  <code class="k">def</code> <code class="nf">__init__</code><code
class="p">(</code><code class="bp">self</code><code class="p">,</code> <code
class="n">year</code><code class="p">):</code>
      <code class="n">Resource</code><code class="o">.</code><code
class="n">__init__</code><code class="p">(</code><code
class="bp">self</code><code class="p">)</code>
      <code class="bp">self</code><code class="o">.</code><code
class="n">year</code> <code class="o">=</code> <code class="n">year</code>
  <code class="k">def</code> <code class="nf">render_GET</code><code
class="p">(</code><code class="bp">self</code><code class="p">,</code> <code
class="n">request</code><code class="p">):</code>
      <code class="k">return</code> <code
class="s">"&lt;html&gt;&lt;body&gt;&lt;pre&gt;</code><code
class="si">%s</code><code
class="s">&lt;/pre&gt;&lt;/body&gt;&lt;/html&gt;"</code> <code
class="o">%</code> <code class="p">(</code><code
class="n">calendar</code><code class="p">(</code><code
class="bp">self</code><code class="o">.</code><code
class="n">year</code><code class="p">),)</code>
```

```
class
```

```
CalendarHome
```

```
(
```

```
Resource
```

```
):
```

```
  <code class="k">def</code> <code class="nf">getChild</code><code
class="p">(</code><code class="bp">self</code><code class="p">,</code> <code
class="n">name</code><code class="p">,</code> <code
class="n">request</code><code class="p">):</code>
      <code class="k">if</code> <code class="n">name</code> <code
class="o">==</code> <code class="s">''</code><code class="p">:</code>
          <code class="k">return</code> <code class="bp">self</code>
      <code class="k">if</code> <code class="n">name</code><code
class="o">.</code><code class="n">isdigit</code><code class="p">():</code>
          <code class="k">return</code> <code class="n">YearPage</code><code
class="p">(</code><code class="nb">int</code><code class="p">(</code><code
class="n">name</code><code class="p">))</code>
      <code class="k">else</code><code class="p">:</code>
          <code class="k">return</code> <code
class="n">NoResource</code><code class="p">()</code>
  <code class="k">def</code> <code class="nf">render_GET</code><code
class="p">(</code><code class="bp">self</code><code class="p">,</code> <code
class="n">request</code><code class="p">):</code>
      <code class="k">return</code> <code
```

```
class="s">"&lt;html&gt;&lt;body&gt;Welcome to the calendar
server!&lt;/body&gt;&lt;/html&gt;"</code>
```

```
root
```

```
=
```

```
CalendarHome
```

```
()
```

```
factory
```

```
=
```

```
Site
```

```
(
```

```
root
```

```
)
```

```
reactor
```

```
.
```

```
listenTCP
```

```
(
```

```
8000
```

```
,
```

```
factory
```

```
)
```

```
reactor
```

```
.
```

```
run
```

```
()
```

</pre>   <div class=„figure-contents mediaobject“><img

src=„https://www.oreilly.com/library/view/twisted-network-programming/9781449326104/httpatomoreillycomsourceoreillyimages1555583.png“ alt=„Calendar“ width=„748“ height=„226“/> Figure 4-2. Calendar This example has the same structure as Example 4-3. A TCP server is started on port 8000, serving the content registered with a

```
Site
```

, which is a subclass of

```
twisted.web.http.HTTPFactory
```

and knows how to manage access to resources. The root resource is

```
CalendarHome
```

, which subclasses

```
Resource
```

to specify how to look up child resources and how to render itself.

```
CalendarHome.getChild
```

describes how to traverse a URL from left to right until we get a renderable resource. If there is no additional component to the requested URL (i.e., the request was for /),

```
CalendarHome
```

returns itself to be rendered by invoking its

```
render_GET
```

method. If the URL has an additional component to its path that is an integer, an instance of

```
YearPage
```

is rendered. If that path component couldn’t be converted to a number, an instance of

```
twisted.web.error.NoResource
```

is returned instead, which will render a generic 404 page. There are a few subtle points to this example that deserve highlighting. <div class=„sect3“ title=„Creating resources that are both renderable and have children“ readability=„49“> <h4 class=„title“ id=„id946146“>Creating resources that are both renderable and have children</h4> Note that

```
CalendarHome
```

does not set

```
isLeaf
```

to

```
True
```

, and yet it is still rendered when we visit http://localhost:8000. In general, only resources that are leaves are rendered; this can be because

```
isLeaf
```

is set to

```
True
```

or because when traversing the resource hierarchy, that resource is where we are when the URL runs out. However, when

```
isLeaf
```

is

```
True
```

for a resource, its

```
getChild
```

method is never called. Thus, for resources that have children,

```
isLeaf
```

cannot be set to

```
True
```

. If we want

```
CalendarHome
```

to both get rendered and have children, we must override its

```
getChild
```

method to dictate resource generation. In

`CalendarHome.getChild`

, if

`name == ''`

(i.e., if we are requesting the root resource), we return ourself to get rendered. Without that

`if`

condition, visiting http://localhost:8000 would produce a 404. Similarly,

`YearPage`

does not have

`isLeaf`

set to

`True`

. That means that when we visit http://localhost:8000/2013, we get a rendered calendar because 2013 is at the end of the URL, but if we visit http://localhost:8000/2013/foo, we get a 404. If we want http://localhost:8000/2013/foo to generate a calendar just like http://localhost:8000/2013, we need to set

`isLeaf`

to

`True`

or have

`YearPage`

override

`getChild`

to return itself, like we do in

`CalendarHome`

.   <h4 class=„title“ id=„id943442“>Redirects</h4> In <a class=„xref“ href=„https://www.oreilly.com/library/view/twisted-network-programming/9781449326104/ch04.html

#twistedadn-CHP-4-EX-6“
title=„Example 4-6. dynamic_dispatch.py“>Example 4-6, visiting
http://localhost:8000 produced a welcome page. What if we wanted
http://localhost:8000 to instead redirect to the calendar for the current year?
In the relevant render method (e.g.,

```
render_GET
```

), instead of rendering the resource at a given URL, we need to construct a redirect with

```
twisted.web.util.redirectTo
```

.

```
redirectTo
```

takes as arguments the URL component to which to redirect, and the request, which still needs to be rendered. <a class=„xref“
href=„https://www.oreilly.com/library/view/twisted-network-programming/9781449326104/ch04.html#twistedadn-CHP-4-EX-7“ title=„Example 4-7. redirectTo“>Example 4-7
shows a revised

```
CalenderHome.render_GET
```

that redirects to the URL for the current year’s calendar (e.g.,
http://localhost:8000/2013) upon requesting the root resource at
http://localhost:8000.
Example 4-7. redirectTo
<pre class=„kn“>from

```
datetime
```

```
import
```

```
datetime
```

```
from
```

```
twisted.web.util
```

```
import
```

```
redirectTo
```

```
def
```

```
render_GET
```

```
(
```

```
self
```

```
,
```

```
request
```

```
):
```

```
  <code class="k">return</code> <code class="n">redirectTo</code><code
class="p">(</code><code class="n">datetime</code><code
class="o">.</code><code class="n">now</code><code class="p">()</code><code
class="o">.</code><code class="n">year</code><code class="p">,</code> <code
class="n">request</code><code class="p">)</code></pre></div>
```

    <div class=„sect1“ title=„Handling POST Requests“ readability=„16“> <h2 class=„title c1“ id=„twistedadn-CHP-4-SECT-3“>Handling POST Requests</h2> To handle POST requests, implement a

```
render_POST
```

method in your

```
Resource
```

. <div class=„sect2“ title=„A Minimal POST Example“ readability=„21“> <h3 class=„title“ id=„twistedadn-CHP-4-SECT-3.2“>A Minimal POST Example</h3> Example 4-8 serves a page where users can fill out and submit to the web server the contents of a text box. The server will then display that text back to the user. Example 4-8. handle_post.py <pre class=„kn“>from

```
twisted.internet
```

```
import
```

```
reactor
```

```
from
```

```
twisted.web.resource
```

```
import
```

```
Resource
```

```
from
```

```
twisted.web.server
```

```
import
```

```
Site
```

```
import
```

```
cgi
```

```
class
```

```
FormPage
```

```
(
```

```
Resource
```

```
):
```

```
  <code class="n">isLeaf</code> <code class="o">=</code> <code
class="bp">True</code>
  <code class="k">def</code> <code class="nf">render_GET</code><code
class="p">(</code><code class="bp">self</code><code class="p">,</code> <code
class="n">request</code><code class="p">):</code>
      <code class="k">return</code> <code class="s">"""</code>
```

```
&lt;html&gt;
```

```
 &lt;body&gt;
```

```
  &lt;form method="POST"&gt;
```

```
   &lt;input name="form-field" type="text" /&gt;
```

```
   &lt;input type="submit" /&gt;
```

```
   &lt;/form&gt;
```

```
   &lt;/body&gt;
```

```
   &lt;/html&gt;
```

```
"""
```

```
  <code class="k">def</code> <code class="nf">render_POST</code><code
class="p">(</code><code class="bp">self</code><code class="p">,</code> <code
```

```
class="n">request</code><code class="p">):</code>
      <code class="k">return</code> <code class="s">"""</code>
```

```
&lt;html&gt;
```

```
 &lt;body&gt;You submitted:
```

```
%s
```

```
&lt;/body&gt;
```

```
 &lt;/html&gt;
```

```
"""
```

```
%
```

```
(
```

```
cgi
```

```
.
```

```
escape
```

```
(
```

```
request
```

```
.
```

```
args
```

```
[
```

```
"form-field"
```

```
][
```

```
0
```

```
]),)
```

```
factory
```

```
=
```

```
Site
```

```
(
FormPage
())
reactor
.
listenTCP
(
8000
,
factory
)
reactor
.
run
()
```

</pre> The

`FormPage`

`Resource`

in handle_post.py implements both

`render_GET`

and

`render_POST`

methods.

`render_GET`

returns the HTML for a blank page with a text box called

```
"form-field"
```

. When a visitor visits http://localhost:8000, she will see this form.

```
render_POST
```

extracts the text inputted by the user from

```
request.args
```

, sanitizes it with

```
cgi.escape
```

, and returns HTML displaying what the user submitted.   <div class=„sect1“ title=„Asynchronous Responses“ readability=„46“> <h2 class=„title c1“ id=„I_sect14_id390469“>Asynchronous Responses</h2> In all of the Twisted web server examples up to this point, we have assumed that the server can instantaneously respond to clients without having to first retrieve an expensive resource (say, from a database query) or do expensive computation. What happens when responding to a request blocks? Example 4-9 implements a dummy

```
BusyPage
```

resource that sleeps for five seconds before returning a response to the request. Example 4-9. blocking.py <pre class=„kn“>from

```
twisted.internet
```

```
import
```

```
reactor
```

```
from
```

```
twisted.web.resource
```

```
import
```

```
Resource
```

```
from
```

```
twisted.web.server
```

```
import
```

```
Site
```

```
import
```

```
time
```

```
class
```

```
BusyPage
```

```
(
```

```
Resource
```

```
):
```

```
  <code class="n">isLeaf</code> <code class="o">=</code> <code
class="bp">True</code>
  <code class="k">def</code> <code class="nf">render_GET</code><code
class="p">(</code><code class="bp">self</code><code class="p">,</code> <code
class="n">request</code><code class="p">):</code>
      <code class="n">time</code><code class="o">.</code><code
class="n">sleep</code><code class="p">(</code><code class="mi">5</code><code
class="p">)</code>
      <code class="k">return</code> <code class="s">"Finally done, at
</code><code class="si">%s</code><code class="s">"</code> <code
class="o">%</code> <code class="p">(</code><code class="n">time</code><code
class="o">.</code><code class="n">asctime</code><code class="p">(),)</code>
```

```
factory
```

```
=
```

```
Site
```

```
(
```

```
BusyPage
```

```
())
```

```
reactor
```

```
.
```

```
listenTCP
```

```
(
```

```
8000
```

```
,
```

```
factory
```

```
)
```

```
reactor
```

```
.
```

```
run
```

```
()
```

</pre> If you run this server and then load http://localhost:8000 in several browser tabs in quick succession, you’ll observe that the last page to load will load N*5 seconds after the first page request, where N is the number of requests to the server. In other words, the requests are processed serially. This is terrible performance! We need our web server to be responding to other requests while an expensive resource is being processed. One of the great properties of this asynchronous framework is that we can achieve the responsiveness that we want without introducing threads by using the

```
Deferred
```

API we already know and love. Example 4-10 demonstrates how to use a

```
Deferred
```

instead of blocking on an expensive resource.

```
deferLater
```

replaces the blocking

```
time.sleep(5)
```

with a

```
Deferred
```

that will fire after five seconds, with a callback to

```
_delayedRender
```

to finish the request when the fake resource becomes available. Then, instead of waiting on that resource,

```
render_GET
```

returns

```
NOT_DONE_YET
```

immediately, freeing up the web server to process other requests. Example 4-10. non_blocking.py <pre class=„kn“>from

```
twisted.internet
```

```
import
```

```
reactor
```

```
from
```

```
twisted.internet.task
```

```
import
```

```
deferLater
```

```
from
```

```
twisted.web.resource
```

```
import
```

```
Resource
```

```
from
```

```
twisted.web.server
```

```
import
```

```
Site
```

```
,
```

```
NOT_DONE_YET
```

```
import
```

```
time
```

```
class
```

```
BusyPage
```

```
(
```

```
Resource
```

```
):
```

```
  <code class="n">isLeaf</code> <code class="o">=</code> <code
class="bp">True</code>
  <code class="k">def</code> <code class="nf">_delayedRender</code><code
class="p">(</code><code class="bp">self</code><code class="p">,</code> <code
class="n">request</code><code class="p">):</code>
      <code class="n">request</code><code class="o">.</code><code
class="n">write</code><code class="p">(</code><code class="s">"Finally done,
at </code><code class="si">%s</code><code class="s">"</code> <code
class="o">%</code> <code class="p">(</code><code class="n">time</code><code
class="o">.</code><code class="n">asctime</code><code class="p">(),))</code>
      <code class="n">request</code><code class="o">.</code><code
class="n">finish</code><code class="p">()</code>
  <code class="k">def</code> <code class="nf">render_GET</code><code
class="p">(</code><code class="bp">self</code><code class="p">,</code> <code
class="n">request</code><code class="p">):</code>
      <code class="n">d</code> <code class="o">=</code> <code
class="n">deferLater</code><code class="p">(</code><code
class="n">reactor</code><code class="p">,</code> <code
class="mi">5</code><code class="p">,</code> <code
class="k">lambda</code><code class="p">:</code> <code
class="n">request</code><code class="p">)</code>
      <code class="n">d</code><code class="o">.</code><code
class="n">addCallback</code><code class="p">(</code><code
class="bp">self</code><code class="o">.</code><code
class="n">_delayedRender</code><code class="p">)</code>
      <code class="k">return</code> <code class="n">NOT_DONE_YET</code>
```

```
factory
```

```
=
```

```
Site
```

```
(
```

```
BusyPage
```

```
())
```

```
reactor
```

```
.
```

```
listenTCP
```

```
(
```

```
8000
```

```
,
```

```
factory
```

```
)
```

```
reactor
```

```
.
```

```
run
```

```
()
```

</pre>   <h3 class=„title“>Tip</h3> If you run Example 4-10 and then load multiple instances of http://localhost:8000 in a browser, you may still find that the requests are processed serially. This is not Twisted’s fault: some browsers, notably Chrome, serialize requests to the same resource. You can verify that the web server isn’t blocking by issuing several simultaneous requests through

```
cURL
```

or a quick Python script.   <div class=„sect1“ title=„More Practice and Next Steps“ readability=„32“> <h2 class=„title c1“ id=„I_sect14_id390474“>More Practice and Next Steps</h2> This chapter introduced Twisted HTTP servers, from the lowest-level APIs up through

```
twisted.web.server
```

. We saw examples of serving static and dynamic content, handling GET and POST requests, and how

to keep our servers responsive with asynchronous responses using

```
Deferred
```

s. The Twisted Web HOWTO index has several in-depth tutorials related to HTTP servers, including on deployment and templating. This page is an excellent series of short, self-contained examples of Twisted Web concepts. The Twisted Web examples directory has a variety of server examples, including examples for proxies, an XML-RPC server, and rendering the output of a server process. Twisted is not a “web framework” like Django, web.py, or Flask. However, one of its many roles is as a framework for building frameworks! An example of this is the Klein micro-web framework, which you can also browse and download at that GitHub page. </section> <section class=„t-bottom-cta bottom-cta bottom-cta-book free-chapter“ readability=„-23“><h2>With Safari, you learn the way you learn best. Get unlimited access to videos, live online training, learning paths, books, interactive tutorials, and more.</h2> </section> </section></section><noscript> </noscript> </html>