simple HTTP server in Java using only Java SE API

Is there a way to create a very basic HTTP server (supporting only GET/POST) in Java using just the Java SE API, without writing code to manually parse HTTP requests and manually format HTTP responses? The Java SE API nicely encapsulates the HTTP client functionality in HttpURLConnection, but is there an analog for HTTP server functionality?

Just to be clear, the problem I have with a lot of ServerSocket examples I've seen online is that they do their own request parsing/response formatting and error handling, which is tedious, error-prone, and not likely to be comprehensive, and I'm trying to avoid it for those reasons.

As an example of the manual HTTP manipulation that I'm trying to avoid:

http://java.sun.com/developer/technicalArticles/Networking/Webserver/WebServercode.html

java http httpserver

edited Sep 17 '10 at 2:34
BalusC
740k 245 2737 2903

asked Sep 17 '10 at 1:29
asker
1,095 2 8 3

3 Umm...the short answer is no. If you want something that handles post and get requests without manually writing the http headers then you could use servlets. But thats java ee. If you don't want to use something like that then sockets and manual parsing is the only other option I know of. – Matt Phillips Sep 17 '10 at 1:32

2 I know this isn't in the spirit of SO, but I would urge you to reconsider you distaste for Java EE API's. As some of the answers have mentioned, there are some very straight-forward implementations such as Jetty that allow you to embed a web server in your stand-alone application while still taking advantage of the servlet api. If you absolutely can't use the Java EE API for some reason than please disregard my comment :-) – Chris Thompson Sep 17 '10 at 3:05

"Servlets" are not really "Java EE". They are just a way of writing plugins that can be called by the surrounding application in response to message activity (these days, generally HTTP requests). Providing a servlet hosting environment "using just the Java SE API" is exactly what Jetty and Tomcat do. Of course you may want to *throw out unwanted complexity* but then you may need to decide on a subset of the allowed attributes and configurations of the GET/POST. It's often not worth it though, except for special security/embedded problems. – David Tonhofer Nov 19 '13 at 17:48

It might be worth going through this list of http servers before making a decision. java-source.net/open-source/web-servers – ThreaT Mar 19 '14 at 10:52

bayou.io – ZhongYu Jun 15 '15 at 17:07

17 Answers

Since Java SE 6, there's a builtin HTTP server in ~~Sun~~ Oracle JRE. The `com.sun.net.httpserver` package summary outlines the involved classes and contains examples.

Here's a kickoff example copypasted from their docs, you can just copy'n'paste'n'run it on Java 6+.

```
package com.stackoverflow.q3732109;

import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpServer;

public class Test {

    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(new InetSocketAddress(8000), 0);
        server.createContext("/test", new MyHandler());
        server.start();
    }

    static class MyHandler implements HttpHandler {
        @Override
        public void handle(HttpExchange t) throws IOException {
            String response = "This is the response";
            t.sendResponseHeaders(200, response.length());
            OutputStream os = t.getResponseBody();
            os.write(response.getBytes());
            os.close();
        }
    }

}
```

Noted should be that the `response.length()` part in their example is bad, it should have been `response.getBytes().length`. Even then, the `getBytes()` method must explicitly specify the charset which you then specify in the response header. Alas, albeit misguiding to starters, it's after all just a basic kickoff example.

Execute it and go to http://localhost:8000/test and you'll see the following response:

> This is the response

As to using `com.sun.*` classes, do note that this is, in contrary to what some developers think, absolutely not forbidden by the well known FAQ Why Developers Should Not Write Programs That Call 'sun' Packages. That FAQ concerns the `sun.*` package (such as `sun.misc.BASE64Encoder`) for internal usage by the Oracle JRE (which would thus kill your application when you run it on a different JRE), not the `com.sun.*` package. Sun/Oracle also just develop software on top of the Java SE API themselves like as every other company such as Apache and so on. Using `com.sun.*` classes is only discouraged (but not forbidden) when it concerns an **implementation** of a certain Java API, such as GlassFish (Java EE impl), Mojarra (JSF impl), Jersey (JAX-RS impl), etc.

edited Mar 27 at 10:02
Gareth Davis
21.9k 11 60 96

answered Sep 17 '10 at 2:34
BalusC
740k 245 2737 2903

4 @Software Monkey - why not? It seems to satisfy asker's requirements. – emory Sep 17 '10 at 8:08

10 @Software: it's not. Even more, it's documented. – BalusC Sep 17 '10 at 11:07

13 @Waldheinz: like as @Software you're confusing `sun.*` with `com.sun.*`. For instance, do you see any documentation of `sun.*` API? Look here: java.sun.com/products/jdk/faq/faq-sun-packages.html Does it tell anything about `com.sun.*`? The `com.sun.*` is just used for their own public software which is not part of Java API. They also develop software on top of Java API, like as every other company. – BalusC Jan 10 '11 at 10:37

5 If you're using Eclipse and get an error like "Access restriction: The type HttpExchange is not accessible due to restriction on required library ...", stackoverflow.com/a/10642163 tells how to disable that access check. – Samuli Kärkkäinen Sep 10 '13 at 9:55

8 FWIW this is also present in OpenJDK. – Jason C Feb 6 '15 at 6:37

Check out NanoHttpd

"NanoHTTPD is a light-weight HTTP server designed for embedding in other applications, released under a Modified BSD licence.

It is being developed at Github and uses Apache Maven for builds & unit testing"

edited Jan 3 at 16:04
Ray Hulha
4,806 3 21 34

answered Sep 17 '10 at 2:29
letronje
4,567 6 33 45

4 One caution: It's likely that NanoHTTPD does not have protection against tree-walking attacks - you should check this if it will be serving on a public address. By this I mean attacks where a request like `GET /../../blahblah http/1.1` is issued and the server walks above the website root and into system file

land, serving files that can be used to compromise or remotely attack the system, like a password file. – Lawrence Dol Sep 17 '10 at 4:51

6 That seems to be fixed. The current version generates a 403 if (uri.startsWith("..") || uri.endsWith("..") || uri.indexOf("../") >= 0). – Lena Schimmel May 4 '12 at 9:23

The *com.sun.net.httpserver* solution is not portable across JREs. Its better to use the official webservices API in *javax.xml.ws* to bootstrap a minimal HTTP server...

```
import java.io._
import javax.xml.ws._
import javax.xml.ws.http._
import javax.xml.transform._
import javax.xml.transform.stream._

@WebServiceProvider
@ServiceMode(value=Service.Mode.PAYLOAD)
class P extends Provider[Source] {
  def invoke(source: Source) = new StreamSource( new StringReader("<p>Hello There!
</p>"));
}

val address = "http://127.0.0.1:8080/"
Endpoint.create(HTTPBinding.HTTP_BINDING, new P()).publish(address)

println("Service running at "+address)
println("Type [CTRL]+[C] to quit!")

Thread.sleep(Long.MaxValue)
```

EDIT: this actually works! The above code looks like Groovy or something. Here is a translation to Java which I tested:

```
import java.io.*;
import javax.xml.ws.*;
import javax.xml.ws.http.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;

@WebServiceProvider
@ServiceMode(value = Service.Mode.PAYLOAD)
public class Server implements Provider<Source> {

    public Source invoke(Source request) {
        return  new StreamSource(new StringReader("<p>Hello There!</p>"));
    }

    public static void main(String[] args) throws InterruptedException {

        String address = "http://127.0.0.1:8080/";
        Endpoint.create(HTTPBinding.HTTP_BINDING, new Server()).publish(address);

        System.out.println("Service running at " + address);
        System.out.println("Type [CTRL]+[C] to quit!");

        Thread.sleep(Long.MAX_VALUE);
    }
}
```

edited Jul 7 '14 at 13:30
Adriaan Koster
11k 2 27 41

answered Jun 8 '14 at 7:03
gruenewa
1,128 6 14

1 +1 for being portable. Too bad you can't set the response content type as it is `text/xml` . – icza Sep 5 '14 at 12:23

1 I think you could do <code>class Server implements Provider<DataSource> { </code> ... and then specify the Content-Type within the DataSource's <code>getContentType()</code> method. Furthermore you can also inject the WebServiceContext: <code> @Resource WebServiceContext ctx;</code> to set other headers and to read request parameters. Unfortunately setting the content-type via WebServiceContext does not work. – gruenewa Sep 12 '14 at 20:09

3 Could you explain why com.sun.net.HttpServer isn't portable across JREs please? – javabeangrinder Nov 19 '14 at 15:56

2 No, I do not think so. It will not work on IBMs Java implementation and maybe also others. And even if it works now, internal APIs are allowed to change. Why not just use the the official API? – gruenewa Nov 26 '14 at 5:58

1 Actually, the original code is scala. – Dmitry Ginzburg May 12 '15 at 12:56

Have a look at the "Jetty" web server Jetty. Superb piece of Open Source software that would seem to meet all your requirments.

If you insist on rolling your own then have a look at the "httpMessage" class.

edited Jul 23 '15 at 13:11
agilob
1,938 12 26

answered Sep 17 '10 at 1:36
James Anderson
22.8k 5 36 65

I think jetty api depends on servlet. – irreputable Sep 17 '10 at 3:00

3 @Irreputable: No, Jetty is a highly modular web server, which has a servlet container as one of it's optional modules. – Lawrence Dol Sep 17 '10 at 4:43

"is ther an analog for server functionality" -- yes its the "servlet" API. The servlet container calls your class after it has parsed the headers, cookies etc. – James Anderson Sep 17 '10 at 8:06

Software Monkey: you can't use it without servlet api. as long as we are using the silly notions of java se/ee, jetty is not se. – irreputable Sep 17 '10 at 14:51

2 Jetty is too big and has too much of a learning curve before actual production usage becomes a possibility. – ThreaT Mar 19 '14 at 10:54

I like this question because this is an area where there's continuous innovation and there's always a need to have a light server especially when talking about embedded servers in small(er) devices. I think answers fall into two broad groups.

1. **Thin-server**: server-up static content with minimal processing, context or session processing.
2. **Small-server**: ostensibly a has many httpD-like server qualities with as small a footprint as you can get away with.

While I might consider HTTP libraries like: Jetty, Apache Http Components, Netty and others to be more like a raw HTTP processing facilities. The labelling is very subjective, and depends on the kinds of thing you've been call-on to deliver for small-sites. I make this distinction in the spirit of the question, particularly the remark about...

- "...without writing code to manually parse HTTP requests and manually format HTTP responses..."

These raw tools let you do that (as described in other answers). They don't really lend themselves to a ready-set-go style of making a light, embedded or mini-server. A mini-server is something that can give you similar functionality to a full-function web server (like say, Tomcat) without bells and whistles, low volume, good performance 99% of the time. A thin-server seems closer to the original phrasing just a bit more than raw perhaps with a limited subset functionality, enough to make you look good 90% of the time. My idea of raw would be makes me look good 75% - 89% of the time without extra design and coding. I think if/when you reach the level of WAR files, we've left the "small" for bonsi servers that looks like everything a big server does smaller.

Thin-server options

- Grizzly
- UniRest (multiple-languages)
- NanoHTTPD (just one file)

Mini-server options:

- Spark Java ... Good things are possible with lots of helper constructs like Filters, Templates, etc.
- MadVoc ... aims to be bonsai and could well be such ;-)

Among the other things to consider, I'd include authentication, validation, internationalisation, using something like FreeMaker or other template tool to render page output. Otherwise managing HTML editing and parameterisation is likely to make working with HTTP look like noughts-n-crosses. Naturally it all depends on how flexible you need to be. If it's a menu-driven FAX machine it can be very simple. The more interactions, the '*thicker*' your framework needs to be. Good question, good luck!

answered Sep 4 '14 at 15:05
will
1,501 1 23 49

Note OP's requirement "using only Java SE API"... – 0xbe5077ed May 29 '15 at 20:38

A very basic web server written in java can be found here http://library.sourcerabbit.com/v/?id=19

answered Jan 18 '14 at 22:47
Nikos
71 1 1

Spark is the simplest, here is a quick start guide:
http://sparkjava.com/

edited Jun 22 '15 at 12:32
Ali Shakiba
8,433 9 49 74

answered Apr 2 '15 at 12:52
Laercio
746 5 14

I can strongly recommend looking into Simple, especially if you don't need Servlet capabilities but simply access to the request/reponse objects. If you need REST you can put Jersey on top of it, if you need to output HTML or similar there's Freemarker. I really love what you can do with this combination, and there is relatively little API to learn.

answered Sep 17 '10 at 18:29
Waldheinz
8,406 1 23 51

+1. I like the ideas behind Simple. However, problems come in when attempting to use HTTPS because Mamba takes away the "embeddable" feature from Simple. – ThreaT Mar 19 '14 at 11:40

It's possible to create an httpserver that provides basic support for J2EE servlets with just the JDK and the servlet api in a just a few lines of code.

I've found this very useful for unit testing servlets, as it starts much faster than other lightweight containers (we use jetty for production).

Most very lightweight httpservers do not provide support for servlets, but we need them, so I thought I'd share.

The below example provides basic servlet support, or throws and UnsupportedOperationException for stuff not yet implemented. It uses the com.sun.net.httpserver.HttpServer for basic http support.

```
import java.io.*;
import java.lang.reflect.*;
import java.net.InetSocketAddress;
import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;

import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpServer;

@SuppressWarnings("deprecation")
public class VerySimpleServletHttpServer {
    HttpServer server;
    private String contextPath;
    private HttpHandler httpHandler;

    public VerySimpleServletHttpServer(String contextPath, HttpServlet servlet) {
        this.contextPath = contextPath;
        httpHandler = new HttpHandlerWithServletSupport(servlet);
    }

    public void start(int port) throws IOException {
        InetSocketAddress inetSocketAddress = new InetSocketAddress(port);
        server = HttpServer.create(inetSocketAddress, 0);
        server.createContext(contextPath, httpHandler);
        server.setExecutor(null);
        server.start();
    }

    public void stop(int secondsDelay) {
        server.stop(secondsDelay);
    }

    public int getServerPort() {
        return server.getAddress().getPort();
    }

}

final class HttpHandlerWithServletSupport implements HttpHandler {

    private HttpServlet servlet;
    private final class RequestWrapper extends HttpServletRequestWrapper {
        private final HttpExchange ex;
        private final Map<String, String[]> postData;
        private final ServletInputStream is;
        private final Map<String, Object> attributes = new HashMap<>();

        private RequestWrapper(HttpServletRequest request, HttpExchange ex,
  Map<String, String[]> postData, ServletInputStream is) {
            super(request);
            this.ex = ex;
            this.postData = postData;
            this.is = is;
        }

        @Override
        public String getHeader(String name) {
            return ex.getRequestHeaders().getFirst(name);
        }

        @Override
        public Enumeration<String> getHeaders(String name) {
            return new Vector<String>(ex.getRequestHeaders().get(name)).elements();
        }

        @Override
        public Enumeration<String> getHeaderNames() {
            return new Vector<String>(ex.getRequestHeaders().keySet()).elements();
        }

        @Override
        public Object getAttribute(String name) {
            return attributes.get(name);
        }

        @Override
        public void setAttribute(String name, Object o) {
            this.attributes.put(name, o);
        }

        @Override
        public Enumeration<String> getAttributeNames() {
            return new Vector<String>(attributes.keySet()).elements();
        }

        @Override
        public String getMethod() {
            return ex.getRequestMethod();
        }

        @Override
        public ServletInputStream getInputStream() throws IOException {
            return is;
        }

        @Override
        public BufferedReader getReader() throws IOException {
            return new BufferedReader(new InputStreamReader(getInputStream()));
        }

        @Override
        public String getPathInfo() {
            return ex.getRequestURI().getPath();
        }

        @Override
        public String getParameter(String name) {
            String[] arr = postData.get(name);
            return arr != null ? (arr.length > 1 ? Arrays.toString(arr) : arr[0]) :
  null;
        }

        @Override
        public Map<String, String[]> getParameterMap() {
            return postData;
        }

        @Override
        public Enumeration<String> getParameterNames() {
            return new Vector<String>(postData.keySet()).elements();
        }
    }

    private final class ResponseWrapper extends HttpServletResponseWrapper {
        final ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        final ServletOutputStream servletOutputStream = new ServletOutputStream() {

            @Override
            public void write(int b) throws IOException {
                outputStream.write(b);
            }
        };

        private final HttpExchange ex;
        private final PrintWriter printWriter;
        private int status = HttpServletResponse.SC_OK;
        private ResponseWrapper(HttpServletResponse response, HttpExchange ex) {
            super(response);
            this.ex = ex;
            printWriter = new PrintWriter(servletOutputStream);
        }

        @Override
        public void setContentType(String type) {
            ex.getResponseHeaders().add("Content-Type", type);
        }

        @Override
        public void setHeader(String name, String value) {
            ex.getResponseHeaders().add(name, value);
        }

        @Override
        public javax.servlet.ServletOutputStream getOutputStream() throws
IOException {
            return servletOutputStream;
        }

        @Override
        public void setContentLength(int len) {
            ex.getResponseHeaders().add("Content-Length", len + "");
        }

        @Override
        public void setStatus(int status) {
            this.status = status;
        }

        @Override
        public void sendError(int sc, String msg) throws IOException {
            this.status = sc;
            if (msg != null) {
                printWriter.write(msg);
            }
        }

        @Override
        public void sendError(int sc) throws IOException {
            sendError(sc, null);
        }

        @Override
        public PrintWriter getWriter() throws IOException {
            return printWriter;
        }

        public void complete() throws IOException {
            try {
                printWriter.flush();
                ex.sendResponseHeaders(status, outputStream.size());
                if (outputStream.size() > 0) {
                    ex.getResponseBody().write(outputStream.toByteArray());
                }
                ex.getResponseBody().flush();
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                ex.close();
            }
        }
    }

    public HttpHandlerWithServletSupport(HttpServlet servlet) {
        this.servlet = servlet;
    }

    @SuppressWarnings("deprecation")
    @Override
    public void handle(final HttpExchange ex) throws IOException {
        byte[] inBytes = getBytes(ex.getRequestBody());
        ex.getRequestBody().close();
        final ByteArrayInputStream newInput = new ByteArrayInputStream(inBytes);
        final ServletInputStream is = new ServletInputStream() {

            @Override
            public int read() throws IOException {
                return newInput.read();
            }
        };

        Map<String, String[]> parsePostData = new HashMap<>();

        try {

parsePostData.putAll(HttpUtils.parseQueryString(ex.getRequestURI().getQuery()));

            // check if any postdata to parse
            parsePostData.putAll(HttpUtils.parsePostData(inBytes.length, is));
        } catch (IllegalArgumentException e) {
            // no postData - just reset inputstream
            newInput.reset();
        }
        final Map<String, String[]> postData = parsePostData;

        RequestWrapper req = new
RequestWrapper(createUnimplementAdapter(HttpServletRequest.class), ex, postData,
is);

        ResponseWrapper resp = new
ResponseWrapper(createUnimplementAdapter(HttpServletResponse.class), ex);

        try {
            servlet.service(req, resp);
            resp.complete();
        } catch (ServletException e) {
            throw new IOException(e);
        }
    }

    private static byte[] getBytes(InputStream in) throws IOException {
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        byte[] buffer = new byte[1024];
        while (true) {
            int r = in.read(buffer);
            if (r == -1)
                break;
            out.write(buffer, 0, r);
        }
        return out.toByteArray();
    }

    @SuppressWarnings("unchecked")
    private static <T> T createUnimplementAdapter(Class<T> httpServletApi) {
        class UnimplementedHandler implements InvocationHandler {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
                throw new UnsupportedOperationException("Not implemented: " +
method + ", args=" + Arrays.toString(args));
            }
        }

        return (T)
Proxy.newProxyInstance(UnimplementedHandler.class.getClassLoader(),
                new Class<?>[] { httpServletApi },
                new UnimplementedHandler());
    }
}
```

answered Nov 28 '13 at 8:55
f.carlsen
306 3 7

You may also have a look at some NIO application framework such as:

1. Netty: http://jboss.org/netty
2. Apache Mina: http://mina.apache.org/ or its subproject AsyncWeb: http://mina.apache.org/asyncweb/

answered Sep 18 '10 at 12:04
ThiamTeck
330 1 6

Once upon a time I was looking for something similar - a lightweight yet fully functional HTTP server that I could easily embed and customize. I found two types of potential solutions:

- Full servers that are not all that lightweight or simple (for an extreme definition of lightweight.)
- Truly lightweight servers that aren't quite HTTP servers, but glorified ServerSocket examples that are not even remotely RFC-compliant and don't support commonly needed basic functionality.

So... I set out to write JLHTTP - The Java Lightweight HTTP Server.

You can embed it in any project as a single (if rather long) source file, or as a ~50K jar (~35K stripped) with no dependencies. It strives to be RFC-compliant and includes extensive documentation and many useful features while keeping bloat to a minimum.

Features include: virtual hosts, file serving from disk, mime type mappings via standard mime.types file, directory index generation, welcome files, support for all HTTP methods, conditional ETags and If-* header support, chunked transfer encoding, gzip/deflate compression, basic HTTPS (as provided by the JVM), partial content (download continuation), multipart/form-data handling for file uploads, multiple context handlers via API or annotations, parameter parsing (query string or multipart/form-data body), etc.

I hope others find it useful :-)

answered Feb 25 '16 at 21:51
amichair
2,642 1 13 11

This code is better than ours, you only need to add 2 libs: **javax.servelet.jar** and **org.mortbay.jetty.jar**.

Class Jetty:

```
package jetty;

import java.util.logging.Level;
import java.util.logging.Logger;
import org.mortbay.http.SocketListener;
import org.mortbay.jetty.Server;
import org.mortbay.jetty.servlet.ServletHttpContext;

public class Jetty {

    public static void main(String[] args) {
        try {
            Server server = new Server();
            SocketListener listener = new SocketListener();

            System.out.println("Max Thread :" + listener.getMaxThreads() + " Min
Thread :" + listener.getMinThreads());

            listener.setHost("localhost");
            listener.setPort(8070);
            listener.setMinThreads(5);
            listener.setMaxThreads(250);
            server.addListener(listener);

            ServletHttpContext context = (ServletHttpContext)
server.getContext("/");
            context.addServlet("/MO", "jetty.HelloWorldServlet");

            server.start();
            server.join();

        /*//We will create our server running at http://localhost:8070
        Server server = new Server();
        server.addListener(":8070");

        //We will deploy our servlet to the server at the path '/'
        //it will be available at http://localhost:8070
        ServletHttpContext context = (ServletHttpContext) server.getContext("/");
        context.addServlet("/MO", "jetty.HelloWorldServlet");

        server.start();
        */

        } catch (Exception ex) {
            Logger.getLogger(Jetty.class.getName()).log(Level.SEVERE, null, ex);
        }

    }
}
```

Servlet class:

```
package jetty;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloWorldServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest httpServletRequest, HttpServletResponse
httpServletResponse) throws ServletException, IOException
    {
        String appid = httpServletRequest.getParameter("appid");
        String conta = httpServletRequest.getParameter("conta");

        System.out.println("Appid : "+appid);
        System.out.println("Conta : "+conta);

        httpServletResponse.setContentType("text/plain");
        PrintWriter out = httpServletResponse.getWriter();
        out.println("Hello World!");
        out.close();
    }
}
```

edited Sep 21 '12 at 10:35
Alex K
16.2k 14 66 88

answered Sep 16 '12 at 11:36
leandro
21 1

2 The question asks for a purely Java SE solution. You'll find that jetty implements the Java EE API. – Sridhar Sep 17 '12 at 15:58

Jetty runs perfectly well using standard Java SE and therefore fits the requirements. It *implements* parts of the Java EE API, it does not *need* it. There is a difference. – David Tonhofer Nov 19 '13 at 17:24

This does not qualify. *"using just the Java SE API"*. `*.Servlet.jar` and `*.jetty.jar` are obviously not part of Java SE. – icza Sep 5 '14 at 12:54

do i need to set jetty up? or can I just unclude those two jars and run this file? – Paul Preibisch Mar 23 '15 at 17:41

---

checkout Simple. its a pretty simple embeddable server with built in support for quite a variety of operations. I particularly love its threading model..

Amazing!

answered Jul 15 '15 at 10:43
Olu
1,043 10 20

Check out `takes`. Look at https://github.com/yegor256/takes for quick info

answered Oct 15 '15 at 14:20
George
1,840 3 14 29

How about Apache Commons HttpCore project?

From the web site:... HttpCore Goals

- Implementation of the most fundamental HTTP transport aspects
- Balance between good performance and the clarity & expressiveness of API
- Small (predictable) memory footprint
- Self contained library (no external dependencies beyond JRE)

answered Feb 11 '13 at 21:52
I. Joseph
301 1 3 13

That is probably too low-level. One should at least aim for a solution that calls your code at the servlet API level, unless one wants to deal with all the concepts like chunking, encoding etc. onself. It can be fun though. – David Tonhofer Nov 19 '13 at 17:35

You can write a pretty simple embedded Jetty Java server.

Embedded Jetty means that the server (Jetty) shipped together with the application as opposed of deploying the application on external Jetty server.

So if in non-embedded approach your webapp built into WAR file which deployed to some external server (Tomcat / Jetty / etc), in embedded Jetty, you write the webapp and instantiate the jetty server in the same code base.

An example for embedded Jetty Java server you can git clone and use: https://github.com/stas-slu/embedded-jetty-java-server-example

answered Jun 29 at 11:14
Stas
2,250 2 18 53

It's possible, because there are so many HTTP servers written in pure Java.

The question is, how come there is no well-known, simple HTTP server in Java (Jetty is not simple enough)? That's a good question. I'm sure if you google "simple java http server", there's a ton of simple implementations. But there is no "famous" one. Maybe Java people mostly do web apps and they've already got a non-simple http server.

A simple yet powerful HTTP server can be implemented in less than 1000 lines of Java.

edited Jun 16 '16 at 2:25
Luc125
4,228 21 37

answered Sep 17 '10 at 3:08
irreputable
35.6k 5 49 75

15 This answer isn't helpful. – Michael Brewer-Davis Sep 17 '10 at 4:40

I disagree. This answer is spot on. I feel the exact same way about searching for a popular yet simple embeddable http solution that does everything it should. – ThreaT Mar 19 '14 at 10:53

protected by Community ♦ Sep 10 '15 at 5:31

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?