JEP 408: Simple Web Server

Owner	Julia Boes
Type	Feature
Scope	JDK
Status	Closed/Delivered
Release	18
Component	core-libs/java.net
Discussion	net dash dev at openjdk dot java dot net
Effort	S
Duration	S
Reviewed by	Alex Buckley, Brian Goetz, Chris Hegarty, Daniel Fuchs
Endorsed by	Brian Goetz
Created	2021/01/27 12:47
Updated	2022/03/07 10:20
Issue	8260510

Summary

Provide a command-line tool to start a minimal web server that serves static files only. No CGI or servlet-like functionality is available. This tool will be useful for prototyping, ad-hoc coding, and testing purposes, particularly in educational contexts.

Goals

- Offer an out-of-the-box static HTTP file server with easy setup and minimal functionality.
- Reduce developer activation energy and make the JDK more approachable.
- Provide a default implementation via the command line together with a small API for programmatic creation and customization.

Non-Goals

- It is not a goal to provide a feature-rich or commercial-grade server. Far better alternatives exist in the form of server frameworks (e.g., Jetty, Netty, and Grizzly) and production servers (e.g., Apache Tomcat, Apache httpd, and NGINX). These full-fledged and performance-optimized technologies take effort to configure, which is exactly what we want to avoid.
- It is not a goal to provide specialty features such as authentication, access control, or encryption. The server is intended solely for testing, development, and debugging. Accordingly, its design is explicitly minimal so as to avoid confusion with a full-featured server application.

Motivation

A common rite of passage for developers is to serve a file on the web, likely a "Hello, world!" HTML file. Most computer science curricula introduce students to web development, where local testing servers are commonly used. Developers usually also learn about system administration and web services, other areas where development tools with basic server functionality can come in handy. Educational and informal tasks such as these are where a small out-of-the-box server is desirable. Use cases include:

- Web development testing, where a local testing server is used to simulate a client-server set up.
- Web-service or application testing, where static files are used as API stubs in a directory structure that mirrors RESTful URLs and contains dummy data.
- Informal browsing and sharing of files across systems to, e.g., search a directory on a remote server from your local machine.

In all these cases we can, of course, use a web-server framework, but that approach has a high activation energy: We have to look for options, pick one, download it, configure it, and figure out how to use it before we can serve our first request. These steps amount to quite a bit of ceremony, which is a drawback; getting stuck somewhere on the way can be frustrating and might even hinder the further use of Java. A basic web server spun up from the command line or via a few lines of code lets us bypass this ceremony, so that we can instead focus on the task at hand.

Python, Ruby, PHP, Erlang, and many other platforms offer out-of-the-box servers run from the command line. This variety of existing alternatives demonstrates a recognized need for this type of tool.

Description

The Simple Web Server is a minimal HTTP server for serving a single directory hierarchy. It is based on the web server implementation in the `com.sun.net.httpserver` package that has been included in the JDK since 2006. The package is officially supported, and we extend it with APIs to simplify server creation and enhance request handling. The Simple Web Server can be used via the dedicated command-line tool `jwebserver` or programmatically via its API.

Command-line tool

The following command starts the Simple Web Server:

```
$ jwebserver
```

If startup is successful then `jwebserver` prints a message to System.out listing the local address and the absolute path of the directory being served. For example:

```
$ jwebserver
Binding to loopback by default. For all interfaces use "-b 0.0.0.0" or "-b ::".
Serving /cwd and subdirectories on 127.0.0.1 port 8000
URL: http://127.0.0.1:8000/
```

By default, the server runs in the foreground and binds to the loopback address and port 8000. This can be changed with the `-b` and `-p` options. For example, to run the server on port 9000, use:

```
$ jwebserver -p 9000
```

For example, to bind the server to all interfaces:

```
$ jwebserver -b 0.0.0.0
Serving /cwd and subdirectories on 0.0.0.0 (all interfaces) port 8000
URL: http://123.456.7.891:8000/
```

By default, files are served from the current directory. A different directory can be specified with the `-d` option.

Only idempotent HEAD and GET requests are served. Any other requests receive a `501 - Not Implemented` or a `405 - Not Allowed` response. GET requests are mapped to the directory being served, as follows:

- If the requested resource is a file, its content is served.
- If the requested resource is a directory that contains an index file, the content of the index file is served.
- Otherwise, the names of all files and subdirectories of the directory are listed. Symbolic links and hidden files are not listed or served.

The Simple Web Server supports HTTP/1.1 only. There is no HTTPS support.

MIME types are configured automatically. For example, `.html` files are served as `text/html` and `.java` files are served as `text/plain`.

By default, every request is logged on the console. The output looks like this:

```
127.0.0.1 - - [10/Feb/2021:14:34:11 +0000] "GET /some/subdirectory/ HTTP/1.1" 200 -
```

Logging output can be changed with the `-o` option. The default setting is `info`. The `verbose` setting additionally includes the request and response headers as well as the absolute path of the requested resource.

Once started successfully, the server runs until it is stopped. On Unix platforms, the server can be stopped by sending it a SIGINT signal (`Ctrl+C` in a terminal window).

The `-h` option displays a help message listing all options, which follow the guidelines in JEP 293. A `jwebserver man` page is also available.

```
Options:
        -h or -? or --help
                Prints the help message and exits.

        -b addr or --bind-address addr
                Specifies the address to bind to.  Default: 127.0.0.1 or ::1 (loopback).  For
                all interfaces use -b 0.0.0.0 or -b ::.

        -d dir or --directory dir
                Specifies the directory to serve.  Default: current directory.

        -o level or --output level
                Specifies the output format.  none | info | verbose.  Default: info.

        -p port or --port port
                Specifies the port to listen on.  Default: 8000.

        -version or --version
                Prints the version information and exits.

        To stop the server, press Ctrl + C.
```

API

While the command-line tool is useful, what if one wants to use the components of the Simple Web Server (i.e., server, handler, and filter) with existing code, or further customize the behavior of the handler? While some configuration is possible on the command line, a concise and intuitive programmatic solution for creation and customization would improve the utility of the server components. To bridge the gap between the simplicity of the command-line tool and the write-it-yourself approach of the current `com.sun.net.httpserver` API, we define new APIs for server creation and customized request handling.

The new classes are `SimpleFileServer`, `HttpHandlers`, and `Request`, each built on existing classes and interfaces in the `com.sun.net.httpserver` package: `HttpServer`, `HttpHandler`, `Filter`, and `HttpExchange`.

The `SimpleFileServer` class supports the creation of a file server, a file-server handler, and an output filter:

```
package com.sun.net.httpserver;

public final class SimpleFileServer {
    public static HttpServer createFileServer(InetSocketAddress addr,
                                               Path rootDirectory,
                                               OutputLevel outputLevel) {...}
    public static HttpHandler createFileHandler(Path rootDirectory) {...}
    public static Filter createOutputFilter(OutputStream out,
                                            OutputLevel outputLevel) {...}
    ...
}
```

With this class, a minimal yet customized server can be started in a few lines of code in `jshell`:

```
jshell> var server = SimpleFileServer.createFileServer(new InetSocketAddress(8000),
   ...> Path.of("/some/path"), OutputLevel.VERBOSE);
jshell> server.start()
```

A customized file-server handler can be added to an existing server:

```
jshell> var server = HttpServer.create(new InetSocketAddress(8080),
   ...> 10, "/store/", new SomePutHandler());
jshell> var handler = SimpleFileServer.createFileHandler(Path.of("/some/path"));
jshell> server.createContext("/browse/", handler);
jshell> server.start();
```

A customized output filter can be added to a server during creation:

```
jshell> var filter = SimpleFileServer.createOutputFilter(System.out,
   ...> OutputLevel.INFO);
jshell> var server = HttpServer.create(new InetSocketAddress(8080),
   ...> 10, "/store/", new SomePutHandler(), filter);
jshell> server.start();
```

The last two examples are enabled by new overloaded `create` methods in the `HttpServer` and `HttpsServer` classes:

```
    public static HttpServer create(InetSocketAddress addr,
                                    int backlog,
                                    String root,
                                    HttpHandler handler,
                                    Filter... filters) throws IOException {...}
```

Enhanced request handling

The core functionality of the Simple Web Server is provided by its handler. To support extending this handler for use with existing code, we introduce a new `HttpHandlers` class with two static methods for handler creation and customization as well as a new method in the `Filter` class for adapting a request:

```
package com.sun.net.httpserver;

public final class HttpHandlers {
    public static HttpHandler handleOrElse(Predicate<Request> handlerTest,
                                           HttpHandler handler,
                                           HttpHandler fallbackHandler) {...}
    public static HttpHandler of(int statusCode, Headers headers, String body) {...}
    {...}
}

public abstract class Filter {
    public static Filter adaptRequest(String description,
                                      UnaryOperator<Request> requestOperator) {...}
    {...}
}
```

`handleOrElse` complements a conditional handler with another handler, while the factory method of lets you create handlers with pre-set response state. The pre-processing filter obtained from `adaptRequest` can be used to inspect and adapt certain properties of a request before handling it. Use cases for these methods include delegating exchanges based on the request method, creating a "canned response" handler that always returns a certain response, or adding a header to all incoming requests.

The existing API captures an HTTP request as part of a request-response pair represented by an instance of the `HttpExchange` class, which describes the full and mutable state of an exchange. Not all of this state is meaningful for handler customization and adaptation. We therefore introduce a simpler `Request` interface to provide a limited view of the immutable request state:

```
public interface Request {
    URI getRequestURI();
    String getRequestMethod();
    Headers getRequestHeaders();
    default Request with(String headerName, List<String> headerValues)
    {...}
}
```

This enables the straightforward customization of an existing handler, for example:

```
jshell> var h = HttpHandlers.handleOrElse(r -> r.getRequestMethod().equals("PUT"),
   ...> new SomePutHandler(), new SomeHandler());
jshell> var f = Filter.adaptRequest("Add foo header", r -> r.with("Foo", List.of("Bar")));
jshell> var s = HttpServer.create(new InetSocketAddress(8080),
   ...> 10, "/", h, f);
jshell> s.start();
```

Alternatives

We considered an alternative for the command-line tool:

- `java -m jdk.httpserver`: Initially, the Simple Web Server was run with the command `java -m jdk.httpserver` rather than with a dedicated command-line tool. While this is still possible (in fact `jwebserver` uses the `java -m ...` command under the hood), we decided to introduce a dedicated tool to improve convenience and approachability.

We considered several API alternatives during prototyping:

- A new class `DelegatingHandler` — Bundle the customization methods in a separate class that implements the `HttpHandler` interface. We discarded this option since it comes at the cost of introducing a new type without adding more functionality. This new type would also be hard to discover. The `HttpHandlers` class, on the other hand, uses the pattern of outboarding, where static helper methods or factories of a class are bundled in a new class. The almost-identical name makes it easy to find the class, facilitates the understanding and use of the new API points, and hides the implementation details of delegation.
- `HttpHandler` as a service — Turn `HttpHandler` into a service and provide an internal file-server handler implementation. The developer could either provide a custom handler or use the default provider. The disadvantage of this approach is that it is more difficult to use and rather elaborate for the small set of functionality we want to provide.
- `Filter` instead of `HttpHandler` — Use only filters, not handlers, to process the request. Filters are typically pre- or post-processing, meaning they access a request either before or after the handler is invoked, for example for authentication or logging. However, they were not designed to fully replace handlers. Using them in this way would be counter-intuitive and the methods would be harder to find.

Testing

The core functionality of the command-line tool is provided by the API, so most of our testing effort will focus on the API. The API points can be tested in isolation with unit tests and the existing test framework. We will focus particularly on file-system access and URI sanitization. We will complement the API tests with command and sanity testing of the command-line tool.

Risks and Assumptions

This simple server is intended for testing, development, and debugging purposes only. Within this scope the general security concerns of servers apply, and will be addressed by following security best practices and thorough testing.