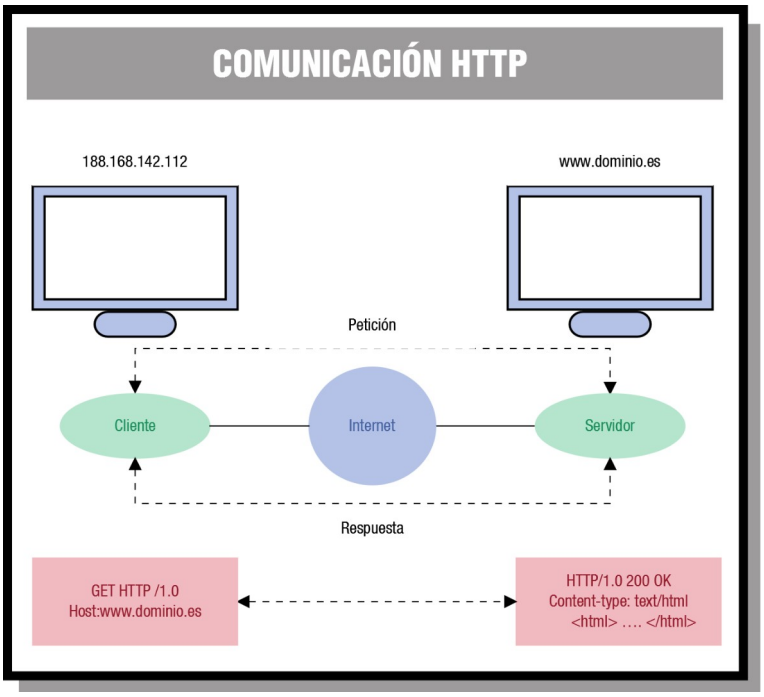
I.E.S Aguadulce

Titulación: Desarrollo de Aplicaciones Multiplataforma

Asignatura: Programación de servicios y procesos



Tarea número 4

– *Generación de servicios en red* –

Autor:	David Jiménez Riscardo
Teléfono:	618882196
E-Mail:	david.jimenez.riscardo@gmail.com

Dos Hermanas, 15 de Febrero del año 2022

Table of Contents

1 CLIENTE HTTP CON URL CONNECTION.....III

2 SERVIDOR HTTP BÁSICO CON HTTPSERVER.....VI

2.1 Clase ServidorHttp básica.....VIII

2.2 Clase HttpHandlerSaludar.....X

3 SERVIDOR HTTP CONCURRENTE CON HTTPSERVER.....XII

3.1 Clase ServidorHttp concurrente.....XV

3.2 Clase HttpHandlerEsPrimo.....XVI

4 FICHAS DE LA TAREA.....XVII

4.1 Ficha 1.....XVII

4.2 Ficha 2.....XX

4.3 Ficha 3.....XXII

5 EVALUACIÓN DE LA TAREA.....XXIV

En esta tarea trabajaremos con conceptos relacionados con los protocolos del nivel de aplicación y la programación de servicios, usando clases para acceder a los recursos de la Web, así como poniendo en práctica nuevamente el concepto de multihilo para poder trabajar de forma concurrente, pero en esta ocasión mediante herramientas que nos permiten llevar a cabo estas tareas desde un nivel mucho más alto, simplificándonos muchísimo el trabajo como programadores. Para ello tendrás que ir realizando cada uno de los ejercicios que se describen en los siguientes apartados así como rellenar la Ficha de la tarea donde explicarás el funcionamiento de cada componente y analizarás algunos escenarios específicos de prueba. La ficha la puedes descargar desde la sección Información de interés.

Para realizar estos programas no tienes que desarrollar ningún entorno gráfico. Serán aplicaciones que se ejecutarán en consola y que pueden recibir parámetros al ser ejecutadas. La única aplicación gráfica que tendrás que usar será un navegador web (Mozilla Firefox, Google Chrome, Microsoft Edge, Opera, Safari, etc.) para poder probar el correcto funcionamiento de los servidores que implementes.

1 Cliente HTTP con URL Connection

Utilizando la clase URLConnection, debes implementar un cliente HTTP que realice las siguientes operaciones:

1. Reciba como parámetro desde la consola una URL de tipo HTTP.
2. Realice una petición HTTP de esa URL.
3. Muestre por pantalla la respuesta HTTP.
4. Y, en función del tipo de respuesta recibida por el servidor, muestre la siguiente información:
    - Si el contenido recibido es de tipo "*text/HTML*", debe mostrar su idioma (charset), el valor de su cookie y la fecha de la última modificación.
    - Si el contenido recibido es de tipo "*image*", debe mostrar su tamaño, tipo y fecha de la última modificación.
    - Si el contenido recibido es de cualquier otro tipo, debe mostrar un mensaje advirtiéndolo.

Si no se recibe ningún parámetro, no se intentará realizar ninguna conexión y el programa finalizará.

Recuerda que puedes averiguar de qué tipo es el contenido recibido por una URLConnection consultando el valor de la cabecera "*Content-Type*".

Ejemplos de peticiones que funcionan correctamente

Aquí tienes un ejemplo de cómo podría quedar la ejecución de tu cliente automático si se le pasa como parámetro la dirección

"*http://www.iesaguadulce.es/centro/index.php/oferta-formativa/formacion-profesional-a-distancia/dam-modalidad-distancia*"

```
------------
Cliente HTTP del alumno XXXXX
Conectándose a la URL
http://www.iesaguadulce.es/centro/index.php/oferta-formativa/formacion-profesional-a-distancia/dam-modalidad-distancia

Respuesta: HTTP/1.1 200 OK

Fecha última modificación: Wed, 12 Jan 2022 10:20:48 GMT
Valor de la cookie: d55b6042d102af43c03b05910d1b865b=3afvscin3suiced5tjne5onog5
Idioma (charset): utf-8
```

Aquí tienes otro ejemplo de ejecución donde se proporciona como parámetro la dirección

"*http://www.iesaguadulce.es/centro/templates/dd_toysshop_34/images/logo_ies_aguadulce.png*" (el logo de nuestro IES).

```
------------
Cliente HTTP del alumno XXXXX
Conectándose a la URL
http://www.iesaguadulce.es/centro/templates/dd_toysshop_34/images/logo_ies_aguadulce.png

Respuesta: HTTP/1.1 200 OK

Fecha última modificación: Mon, 04 Nov 2019 19:56:30 GMT
Tamaño de imagen: 347124 bytes
Tipo de imagen: png
```

Por último, aquí tienes un ejemplo de petición a un PDF:

"*http://www.iesaguadulce.es/centro/images/Documentos_oficiales/PlanesDeCentro/202021/planes2021_22/ProyectoFormacinProfesionalaDistanciav1_01.pdf*":

```
------------
Cliente HTTP del alumno XXXXX
Conectándose a la URL
http://www.iesaguadulce.es/centro/images/Documentos_oficiales/PlanesDeCentro/
202021/planes2021_22/ProyectoFormacinProfesionalaDistanciav1_01.pdf

Respuesta: HTTP/1.1 200 OK

No se trata de ningún archivo
```

Ejemplos de peticiones que producen error

Ejemplo de una petición con una URL no válida (malformada):

```
CLIENTE HTTP
------------
Cliente HTTP del alumno XXXXX
Conectándose a la URL httx://help.me

Error. URL no válida: unknown protocol: httx
```

Ejemplo de una petición que produce un error de E/S (probamos a desconectar la red para no tener acceso a Internet):

```
CLIENTE HTTP
------------
Cliente HTTP del alumno XXXXX
Conectándose a la URL http://www.google.es

Error de E/S: www.google.es.
```

Ejemplo de una petición a un host inexistente:

```
CLIENTE HTTP
------------
Cliente HTTP del alumno XXXXX
Conectándose a la URL http://www.hijk22.com/

Error de E/S: www.hijk22.com.
```

2 Servidor HTTP básico con HttpServer

Tendrás que implementar un servidor HTTP que escuchará desde un puerto que habrá sido recibido desde la consola como parámetro al iniciarse el programa. Si no se introdujo ningún parámetro o éste fuera inválido, se intentará instalar en el puerto HTTP por omisión (el estándar 80). Si no es posible escuchar por el puerto especificado (porque, por ejemplo, esté ya ocupado o bien porque no se tengan permisos), entonces el programa debe finalizar ordenadamente indicando que el puerto no está disponible. Todos estos posibles errores deben estar bien controlados y debemos evitar que el programa "aborte" abruptamente.

Este servidor HTTP será capaz de atender a peticiones de tipo

"*/saludar?nombre=xxx&apellido=yyy"*.

Por ejemplo escribiendo desde la barra del navegador:

```
http://localhost/saludar?nombre=Diosdado&apellido=Sanchez
```

y devuelva el texto "Hola xxx yyy". Para el ejemplo anterior mostraría en el navegador la siguiente respuesta:

```
Hola Diosdado Sanchez
```

Las clases **HttpServer** y **HttpHandler**, disponibles en el paquete **com.sun.net.httpserver**, nos proporcionan la posibilidad de organizar desde un nivel más alto toda la gestión que debe llevar a cabo un servidor HTTP sin tener que preocuparnos de detalles como abrir/cerrar sockets o tener que estar pendientes de leer de un flujo de entrada (stream).

Utilizando estas herramientas de alto nivel podemos implementar con bastante facilidad un servidor HTTP que nos permitirá concentrarnos en el servicio que queremos proporcionar sin tener que programar los detalles más técnicos de las conexiones. Es decir, que podremos centrarnos más en el nivel de aplicación dejando los detalles del **nivel de transporte** (como por ejemplo la gestión de los sockets) a esas clases especializadas en HTTP.

Para desarrollar nuestro servidor HTTP implementaremos las siguientes clases:

1. **`ServidorHttp`**, programa principal (con método **main**).
2. **`HandlerSaludar`**, gestor o manejador ("handler") del contexto o aplicación web "*/saludar*".

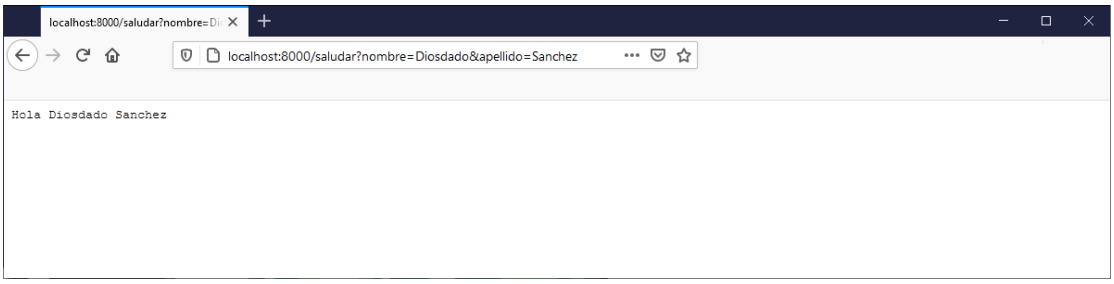
Ejemplos de funcionamiento

Aquí tienes algunas muestras de lo que podría mostrar un cliente (navegador web) según las peticiones que se le envíen al servidor.

Suponemos que el servidor ha sido configurado para escuchar por el **puerto 8000**.

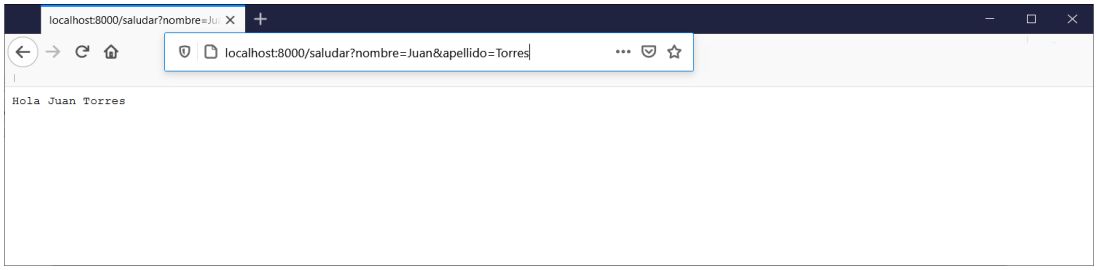
1. Para la petición

   http://localhost:8000/saludar?nombre=Diosdado&apellido=Sanchez



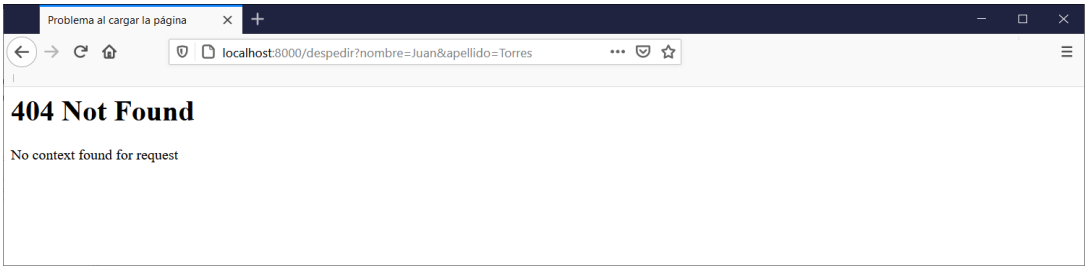
2. Para la petición

   http://localhost:8000/saludar?nombre=Juan&apellido=Torres



3. Para la petición

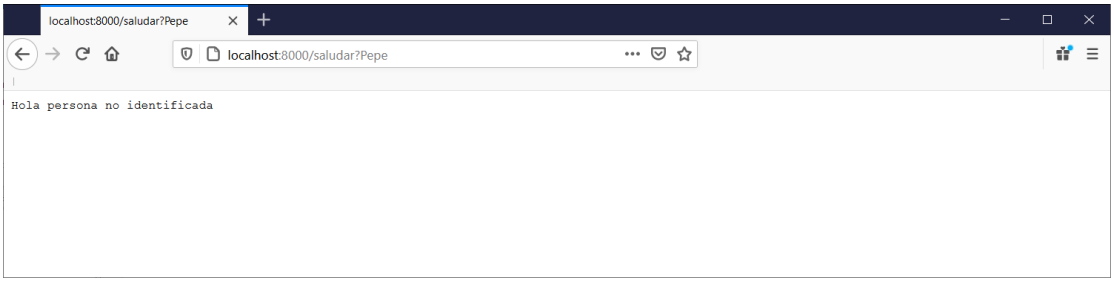
   http://localhost:8000/despedir?nombre=Juan&apellido=Torres



En este caso se está pidiendo un recurso que el servidor no reconoce *("/despedir")*.

4. Para la petición

   http://localhost:8000/saludar?Juan



En este caso se pide un recurso conocido ("*/saludar*"), pero los parámetros no son los esperados (nombre y apellido con el formao "*nombre=xxx&apellido=xxx*").

Mientras tanto, en la consola del servidor podríamos haber observado un comportamiento similar al siguiente:

```
SERVIDOR HTTP
-------------
[06/01/2021 08:20:30:888] Servidor HTTP iniciado en el puerto 8000
[06/01/2021 08:20:39:022] Atendiendo a petición: /saludar?
nombre=Diosdado&apellido=Sanchez
[06/01/2021 08:20:39:023] Respuesta a la petición: /saludar?
nombre=Diosdado&apellido=Sanchez -> Hola Diosdado Sanchez
[06/01/2021 08:20:49:985] Atendiendo a petición: /saludar?
nombre=Juan&apellido=Torres
[06/01/2021 08:20:49:986] Respuesta a la petición: /saludar?
nombre=Juan&apellido=Torres -> Hola Juan Torres
[06/01/2021 08:21:04:999] Atendiendo a petición: /saludar?Juan
[06/01/2021 08:21:04:999] Respuesta a la petición: /saludar?Juan -> Hola persona no
identificada
```

Puedes observar que la petición número 3 no llega ni a ser procesada, pues no cumplía con el contexto *"/saludar"* y por tanto ni siquiera llega ha llegado a ejecutarse el manejador (`HandlerSaludar`). Nuestro servidor se encargará automáticamente de generar un error de tipo 404.

2.1 Clase ServidorHttp básica

La clase `ServidorHttp` será nuestro **programa principal**, que podrá recibir un parámetro desde la consola: el **puerto de escucha**. Si no se recibe ningún parámetro o

ese parámetro no puede ser interpretado como un puerto válido, se utilizará el **puerto de escucha por omisión** para el protocolo HTTP.

A continuación se encargará de crear un objeto `HttpServer` donde se le indicará el **puerto de escucha** y un parámetro llamado `backlog` que nosotros por el momento podremos a 0. La clase `HttpServer` **no dispone de constructores públicos**, por lo que para obtener una instancia de un objeto de este tipo tendrás que usar un método "fábrica" llamado `create`:

```
public static HttpServer create(InetSocketAddress addr, int backlog) throws
IOException
```

En nuestro programa podríamos hacer algo como:

```
httpServer = HttpServer.create(new InetSocketAddress(puerto), 0);
```

Una vez que dispongamos de una instancia de un objeto `HttpServer`, le podremos asignar uno o varios "contextos" mediante el método `createContext`.

```
public abstract HttpContext createContext(String path, HttpHandler handler)
```

Cuando un servidor recibe una petición HTTP se analizará qué "contexto" se está solicitando y si coincide con alguno de los que la aplicación es capaz de interpretar y gestionar, entonces se ejecutará su manejador ("handler") correspondiente. Por ejemplo, en un servidor web podríamos tener las siguientes peticiones válidas:

- "*/*"
- "*/aplicaciones/sumar*"
- "*/aplicaciones/saludar*"
- "*/fecha*"
- "*/apps/buscar*"
- "*/apps/sumar*"
- "*/saludar*"

A cada una de esas posibles cadenas que el servidor puede recibir como petición (por ejemplo "*http://localhost/**aplicaciones/saludar***") se les conoce como "contextos" y tendremos que implementar tantos manejadores como contextos queramos gestionar.

Puedes observar que sobre la cadena de "contexto" luego se podrán incluir parámetros añadiéndole el carácter interrogación ('*?*') seguido de un conjunto parámetros con la estructura ("*nombreParametro=valorParametro*") y separados por el carácter "*ampersand*" ('*&*'). Por ejemplo: "*?nombre=Diosdado&apellido=Sanchez*".

Un ejemplo completo de un "contexto" con parámetros podría ser: "*/saludar?nombre=Diosdado&apellido=Sanchez*" y la URL completa que se escribiría en el cliente (navegador web) podría ser algo así como:

```
httpServer.createContext("/saludar", new HttpHandlerSaludar());
```

Esto implica que tendremos que implementar una clase llamada `HttpHandlerSaludar`, que será quien se encargue de gestionar todas las peticiones al servidor que comiencen por "*/saludar*".

Si la petición HTTP no coincide con ningún contexto de los que hayamos registrado (por ejemplo es "*/despedir*") entonces nuestro objeto instancia de `HttpServer` se encargará automáticamente devolver un mensaje de no encontrado (código de error HTTP 404) sin que tengamos que implementarlo nosotros.

Por último, una vez que tengamos asignado el contexto, solo faltaría lanzar nuestro servidor con el método `start`:

```
httpServer.start();
```

A partir de ahí, se lanzará un hilo independiente para gestionar las peticiones HTTP y nuestro hilo principal podrá seguir realizando otras funciones (o incluso finalizar).

Te recomendamos que le eches un vistazo a la documentación sobre la clase `HttpServer` y especialmente al apartado "*Mapping request URIs to HttpContext paths*".

2.2 Clase HttpHandlerSaludar

Para cada contexto que queramos establecer en nuestro servidor habrá que escribir una clase "manejadora de contexto". Para ello es necesario implementar la interfaz `HttpHandler`. Por ejemplo, para nuestro contexto "*/saludar*" podríamos codificar una clase llamada `HttpHandlerSaludar`. Por el hecho de implementar la interfaz `HttpHandler`, tendremos obligatoriamente que escribir el código el método `handle`:

```
public void handle(HttpExchange exchange) throws IOException
```

En este método habrá que obtener la petición HTTP que se nos ha solicitado (todo lo que se escribiera en la petición del navegador a partir de la barra '/'). Eso podemos hacerlo aplicando el método `getRequestURI()` al objeto de tipo `HttpEx-`

`change` que se recibe como parámetro. A partir de ahí obtendremos un objeto de tipo `URI`:

```
URI uri = exchange.getRequestURI();
```

Esa `URI` podemos analizarla y generar una respuesta adecuada, por ejemplo un saludo utilizando los parámetros que contiene (nombre y apellido). Para analizar y obtener los distintos componentes de una URI dispones de una gran variedad de métodos. En este caso te recomendamos que utilices directamente `getQuery`, que te devolvería una cadena con todo lo que hubiera a partir de la interrogación, es decir, algo así como "*nombre=xxx&apellido=yyy*".

Esa petición podemos mostrarla por pantalla para que tengamos una cierta idea de lo que está haciendo en ese momento nuestro servidor. Podríamos mostrar una salida del tipo:

```
[<Fecha> <Hora>] Atendiendo a petición: <petición>
```

donde <Fecha> <Hora> serían la fecha y hora actual y <petición> la cadena de caracteres que hayamos obtenido con `getQuery`. Si por ejemplo se hubiera realizado la petición "*/saludar?nombre=Diosdado&apellido=Sanchez*", la salida por pantalla podría haber sido:

```
[06/01/2021 08:20:39:022] Atendiendo a petición: /saludar?
nombre=Diosdado&apellido=Sanchez
```

Para obtener la fecha y la hora actual se te proporciona la clase `Utilidades` que contiene el método estático`Utilidades.getFechaHoraActualFormateada`, que devuelve directamente un `String` con la fecha y hora en ese formato.

Si la "query" ("consulta" o "petición") obtenida tiene en efecto el formato "*nombre=xxx&apellido=yyy*", entonces podrías devolver el texto "Hola, xxx yyy". Para ello bastaría con que:

1. envíes el estado "OK" de HTTP como respuesta a la petición;
2. envíes tu texto como parte de tu respuesta a la petición.

Si no coincide con ese patrón, siempre puedes generar un texto de otro tipo. En nuestro caso podríamos decir "*Hola persona no identificada*".

El mecanismo para enviar respuestas está implementado mediante el objeto `HttpExchange` que se recibe como parámetro en el método.

Para en devolver el estado "OK" (o cualquier otro) puedes usar el método `sendResponseHeaders` de la clase `HttpExchange` haciendo algo como:

```
exchange.sendResponseHeaders(200, respuesta.getBytes().length);
```

donde `respuesta` sería el objeto `String` que contendría el texto que deseamos enviar como respuesta.

Finalmente, habría que escribir ese texto de respuesta en el stream o flujo de salida que nos proporciona el objeto `HttpExchange`. Por ejemplo:

```
OutputStream os = exchange.getResponseBody();
os.write(response.getBytes());
os.close();
```

También estaría bien que mostraras por la consola esa respuesta para poder realizar un correcto seguimiento del funcionamiento del servidor. Podrías escribir en pantalla una línea del tipo:

```
[<Fecha> <Hora>] Respuesta a la petición: <petición> -> <respuesta>
```

donde <Fecha> <Hora> serían la fecha y hora actual, <petición> la petición a la que se está dando una contestación y <respuesta> la cadena de caracteres que hayamos generado como respuesta y que vamos a devolver al cliente. Si por ejemplo la respuesta fuera "*Hola Diosdado Sanchez Sanchez*", la salida por pantalla podría haber sido:

```
[06/01/2021 08:20:39:023] Respuesta a la petición: /saludar?
nombre=Diosdado&apellido=Sanchez -> Hola Diosdado Sanchez Sanchez
```

De esa manera finalizaríamos nuestro método `handle` que gestionaría el "contexto" "*/saludar*".

Te recomendamos que le eches un vistazo a la documentación sobre la interfaz `HttpHandler` y las clases `HttpExchange` y `URI`.

3 Servidor HTTP concurrente con HttpServer

Una vez que hayas logrado implementar el servidor HTTP anterior, ahora se trata de darle algo más de funcionalidad. Para ello vamos a:

1. proporcionar dos contextos diferentes: "***/saludar***" y "***/primo***";
2. hacer que el servidor sea concurrente para poder atender a varios clientes simultáneamente.

En el primer caso se trata simplemente de que implementes una nueva clase `HttpHandlerEsPrimo` para dar respuesta a esa nueva funcionalidad que va a ofrecer nuestro servidor.

En el segundo caso podrías:

1. crear tu propio grupo o **conjunto de hilos** (***Thread Pool***) para gestionar la concurrencia;
2. usar el servicio integrado *Thread Pools* que ya incorpora Java (`ExecutorService`).

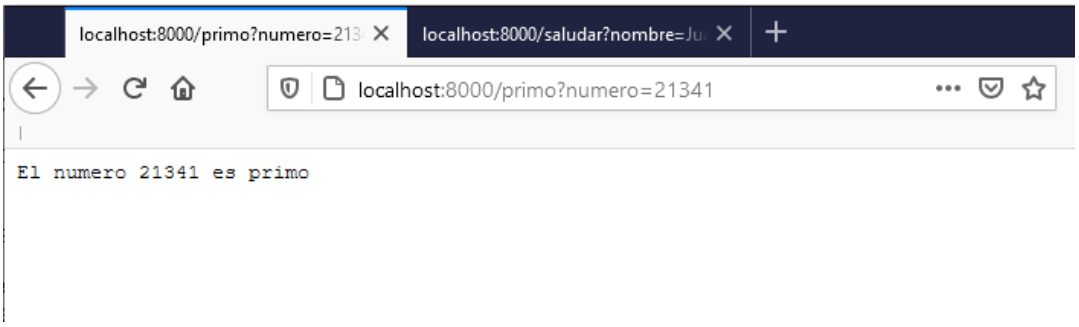
El primer acercamiento sería similar a como ya trabajaste en la tarea anterior, aunque sistematizando un poco más el control de los hilos. Para esta tarea te proponemos el segundo acercamiento, que gestionará automáticamente los hilos por ti facilitándote muchísimo más la labor, que es de lo que se trata, de hacer las cosas más fáciles y no más difíciles evitando tener que "*reinventar la rueda*".

Ejemplos de funcionamiento

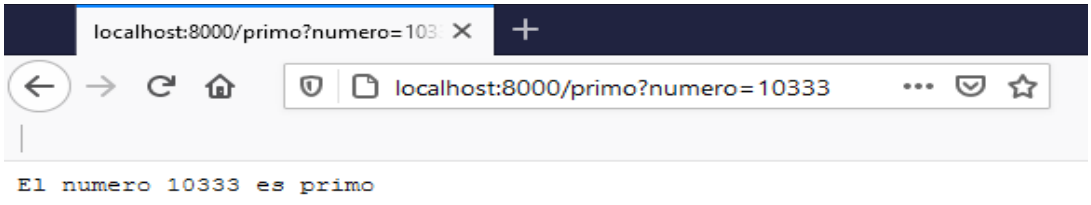
Aquí tienes algunas muestras de lo que podría mostrar un cliente (navegador web) según las peticiones que se le envíen al servidor. En este caso, podría haber algunas peticiones resolviéndose (la primera y la segunda) mientras se responde de manera simultánea a otras más rápidas (la tercera y la cuarta).

Suponemos que el servidor ha sido configurado para escuchar por el **puerto 8000**.

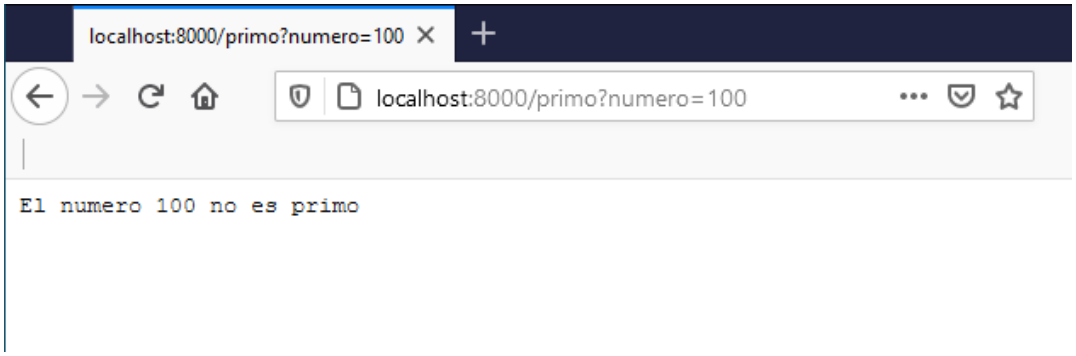
1. Para la petición *http://localhost:8000/primo?numero=21341*



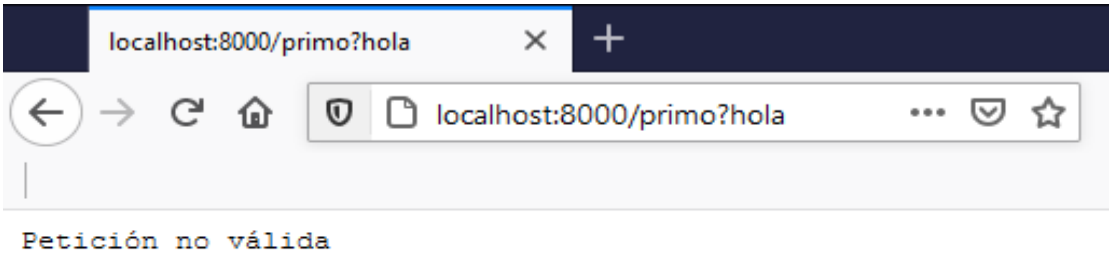
2. Para la petición *http://localhost:8000/primo?numero=10333*



3. Para la petición *http://localhost:8000/primo?numero=100*



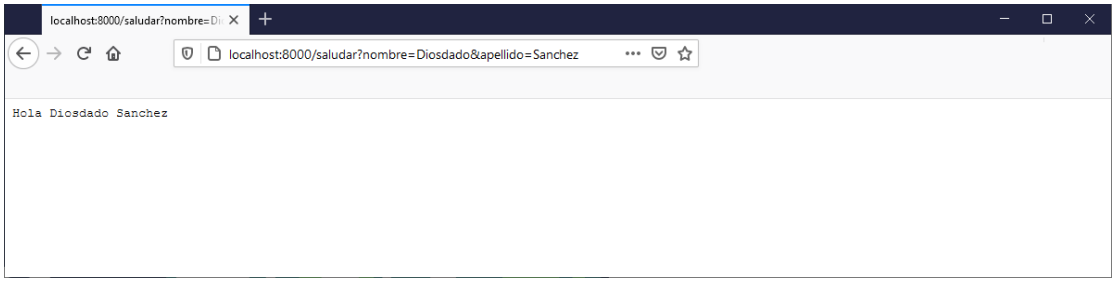
4. Para la petición *http://localhost:8000/primo?hola*



En este caso se pide un recurso conocido ("*/primo*"), pero los parámetros no son los esperados o no se cumple el formato especificado.

5. Para la petición

   http://localhost:8000/saludar?nombre=Diosdado&apellido=Sanchez, que ya tratamos en el anterior servidor, se seguirá obteniendo lo mismo:



Mientras tanto, en la consola del servidor podríamos haber observado un comportamiento similar al siguiente:

```
SERVIDOR HTTP
-------------
[28/01/2021 05:39:21:520] Servidor HTTP iniciado en el puerto 8000
[28/01/2021 05:39:27:492] Atendiendo a petición: /primo?numero=21341
[28/01/2021 05:39:31:638] Atendiendo a petición: /primo?numero=10333
[28/01/2021 05:39:36:754] Atendiendo a petición: /primo?numero=100
[28/01/2021 05:39:36:758] Respuesta a la petición: /primo?numero=100 -> El número
100 no es primo
[28/01/2021 05:39:40:123] Atendiendo a petición: /primo?hola
[28/01/2021 05:39:40:124] Respuesta a la petición: /primo?hola -> Petición no válida
[28/01/2021 05:39:42:653] Atendiendo a petición: /saludar?
nombre=Diosdado&apellido=Sanchez
[28/01/2021 05:39:42:655] Respuesta a la petición: /saludar?
nombre=Diosdado&apellido=Sanchez -> Hola Diosdado Sanchez
[28/01/2021 05:39:47:027] Respuesta a la petición: /primo?numero=10333 -> El número
10333 es primo
[28/01/2021 05:39:57:487] Respuesta a la petición: /primo?numero=21341 -> El número
21341 es primo
```

Las dos primeras peticiones, que son las que más procesamiento van a necesitar (sobre todo la primera), serán las últimas en resolverse.

3.1 Clase ServidorHttp concurrente

En este caso, una vez que dispongamos de una instancia de un objeto `HttpServer`, tendrás que asignarle dos "contextos" mediante el uso del método `createContext`.

```
httpServer.createContext("/saludar", new HttpHandlerSaludar());
httpServer.createContext("/primo",   new HttpHandlerEsPrimo());
```

Esto implica que tendrás que implementar una nueva clase `HttpHandlerEsPrimo` para dar respuesta a las peticiones que se reciban en este contexto.

Una vez que tengas los contextos asignados al objeto `httpServer`, y antes de lanzarlo con el método `start`, tendrás que incorporar la parte de **gestión multihilo** mediante la asignación de un `Executor`.

Para asignar un ejecutor tendremos que usar el método `setExecutor` de la clase `HttpServer`:

```
public abstract void setExecutor(Executor executor)
```

El ejecutor (`Executor`) debe ser establecido antes de llamar al método `start`. Todas las peticiones HTTP que reciba el servidor serán gestionadas mediante tareas asignadas al `Executor`. Si no se llama a este método antes de `start` o si se invoca con un `Executor` nulo, entonces se usa una implementación predeterminada y nuestro servidor será monohilo.

Para nuestro programa podríamos crear un grupo de hilos usando `Executors.newCachedThreadPool()`. Bastaría con que hiciéramos:

```
httpServer.setExecutor(Executors.newCachedThreadPool());
```

Con eso habremos indicado a nuestro servidor HTTP que deseamos que se lance un nuevo hilo para servir a cada petición que se reciba. Esa acción se realizará automáticamente sin que nosotros tengamos que escribir más código al respecto.

Una vez establecido el ejecutor ya podremos lanzar nuestro servidor del mismo modo que hicimos en el ejercicio anterior mediante el método `start`:

```
httpServer.start();
```

A partir de ahí, se lanzará un hilo independiente para gestionar las peticiones HTTP y nuestro hilo principal podrá seguir realizando otras funciones (o incluso finalizar).

3.2 Clase HttpHandlerEsPrimo

Del mismo modo que implementamos la clase `HttpHandlerSaludar` para gestionar contexto "*/saludar*", también tendremos que implementar una nueva clase `HttpHandlerEsPrimo` para gestionar el contexto "*/primo*". Para ello tendremos que escribir el código de su método `handle`.

En el método `handle`, lo primero que tendremos que hacer será nuevamente obtener la cadena con todo lo que haya a partir de la interrogación. En este caso será algo con

el formato "*numero=xxx*" donde xxx será el número cuyo test de primalidad queremos realizar.

Para llevar a cabo esa comprobación de si un número es primo o no, dispondrás de la clase `Utilidades` la cual proporciona el método `esPrimo`, que devuelve `true` si el número es primo y `false` si no lo es. Este método tardará más en ejecutarse cuanto más grande sea su divisor más pequeño. Si el número es primo, tardará bastantes segundos (incluso minutos si el número es grande). En función de lo que nos devuelva ese método, se generará una salida con un texto del tipo "*El número xxx es primo*" o bien del tipo "*El número xxx no es primo*".

Por ejemplo, si la petición fuera "*/primo?numero=21341*" el texto que debería mostrar el cliente sería "*El número 21341 es primo*". Si la petición fuera "*/primo?numero=100*" el texto sería "*El número 100 no es primo*".

El hecho de que se pueda tardar bastante en dar una respuesta nos servirá para probar si funciona correctamente la concurrencia, pues mientras se nos contesta a una llamada a "*/primo*" complicada, que va a tardar un tiempo en ser respondida, podemos comprobar que otra llamada a "*/primo*" más sencilla o a "*/saludar*" es contestada inmediatamente por otro hilo del servidor.

Si la petición recibida no respeta el formato "/primo?*numero=xxx*", en lugar de responder con un texto del tipo "*El número xxx es primo*" o "*El número xxx no es primo*", se podrá responder con un texto del tipo "*Petición no válida*" o algo similar.

4 Fichas de la tarea

4.1 Ficha 1

Incluir breve explicación del funcionamiento del programa y capturas del funcionamiento de tu cliente interactivo utilizando los ejemplos descritos en el apartado 1 (tres ejemplos que funcionan: en las web del IES Aguadulce, y tres que producen errores).

En las capturas debe aparecer la salida por consola de cliente, donde se pueda observar cómo ha funcionado la petición: si ha funcionado o no, qué tipo de respuesta se ha obtenido y la información solicitada para cada tipo de respuesta.

El programa cliente recibe una URL por la entrada estándar introducida por el usuario para posteriormente realizar una petición HTTP de esa URL.

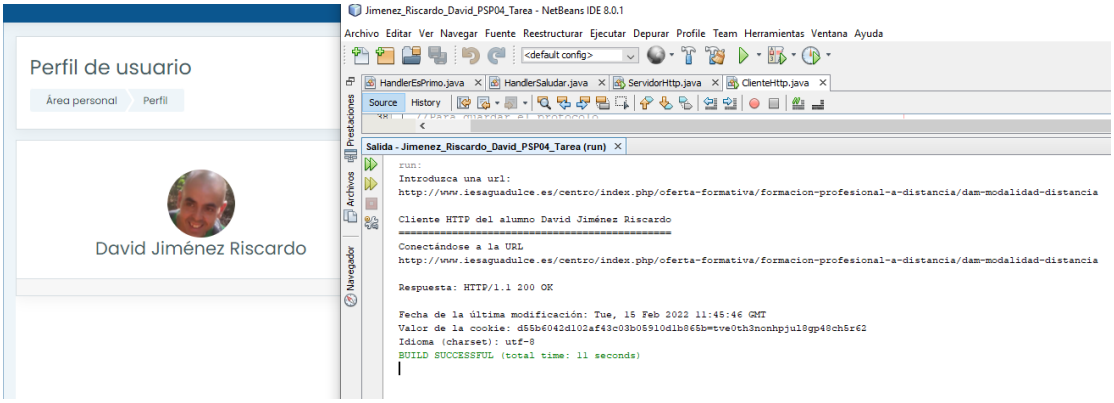
Para terminar mostraremos por consola la respuesta HTTP. En función de la respuesta recibida del servidor mostraremos por consola diferente información al respecto.

Durante la realización del ejercicio he observado que la función “getProtocol” de la clase URL, no devuelve el protocolo si este no existe por lo que he desarrollado una función para el caso en el que se introduzca el protocolo de forma incorrecta.

A continuación se facilitan una serie de ejemplos.

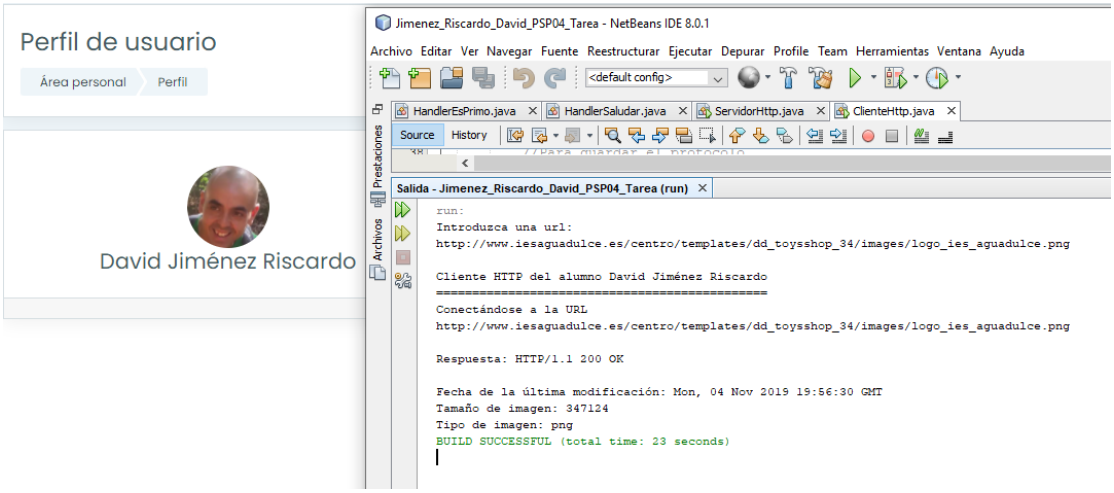
Ejemplo 1, petición a una página web

http://www.iesaguadulce.es/centro/index.php/oferta-formativa/formacion-profesional-a-distancia/dam-modalidad-distancia



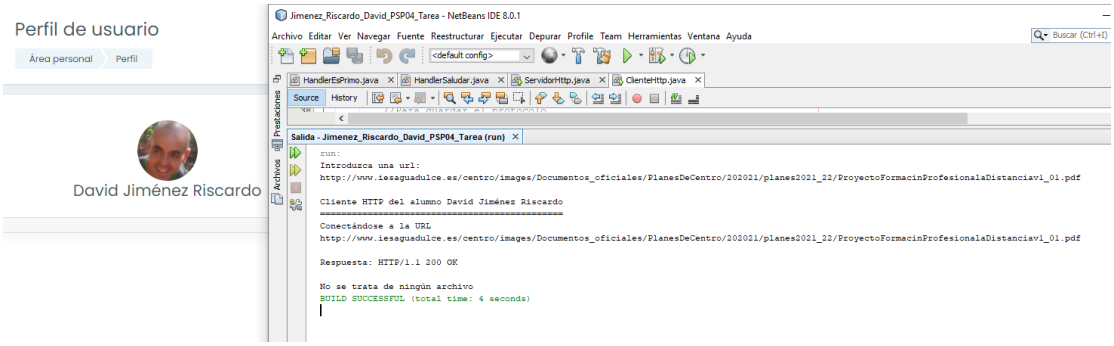
Ejemplo 2, petición a una imagen de una página web

http://www.iesaguadulce.es/centro/templates/dd_toysshop_34/images/logo_ies_aguadulce.png



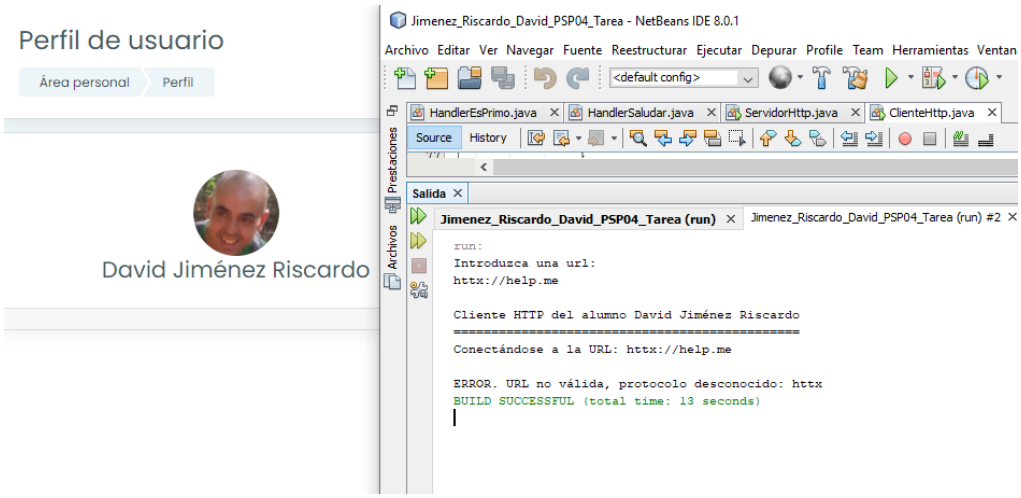
Ejemplo 3, petición a un documento PDF

http://www.iesaguadulce.es/centro/images/Documentos_oficiales/PlanesDeCentro/202021/planes2021_22/ProyectoFormacinProfesionalaDistanciav1_01.pdf



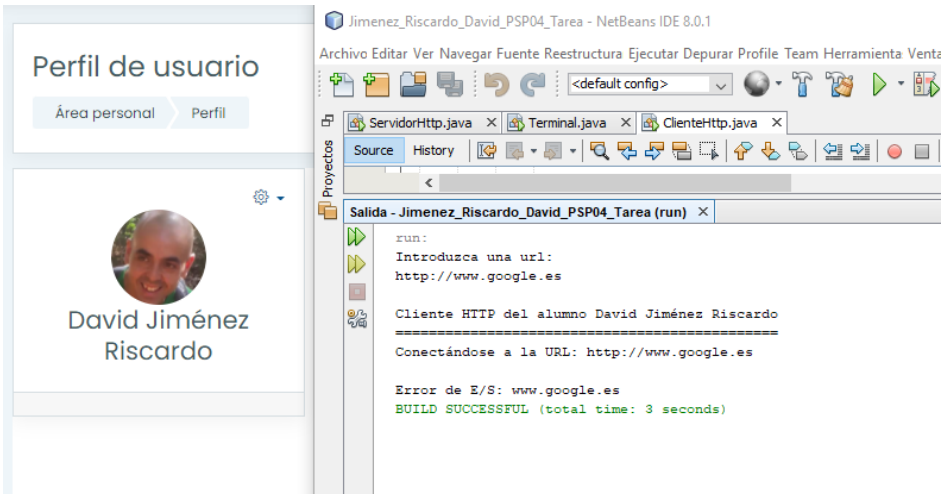
Ejemplo 4, petición a una URL no válida

httx://help.me



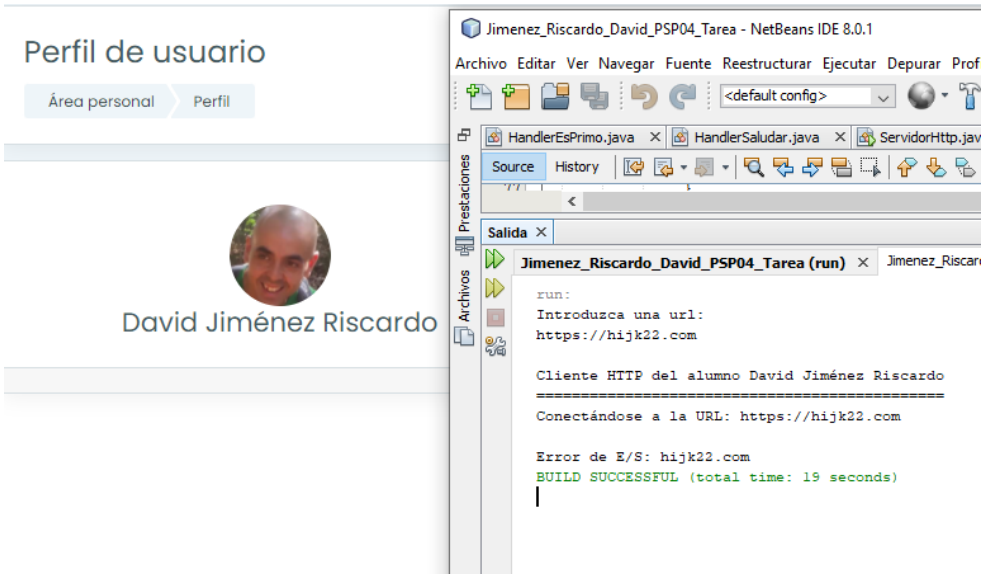
Ejemplo 5, petición a una URL que produce un error de E/S (sin acceso a Internet)

http://www.google.es



Ejemplo 6, petición a una URL que no existe

http://hijk22.com



4.2 Ficha 2

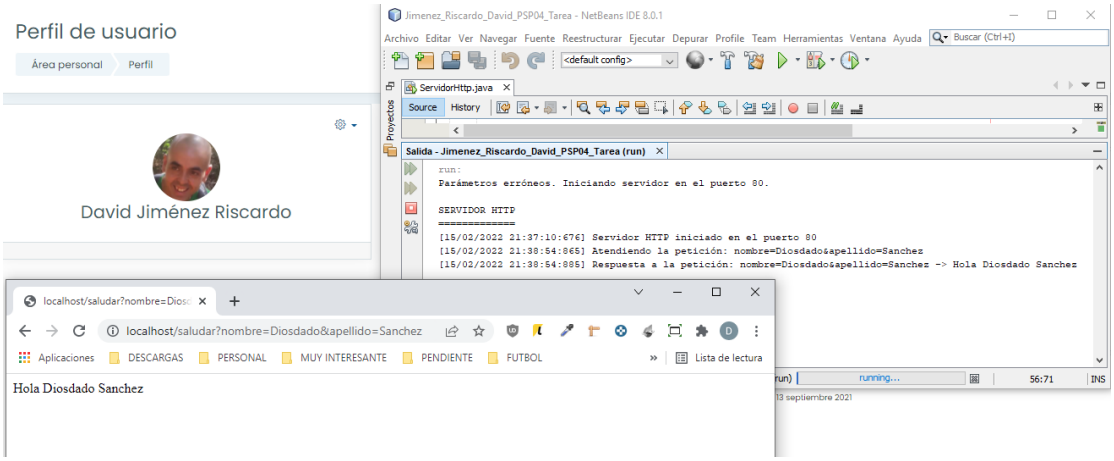
Incluir breve explicación del funcionamiento del programa y capturas del funcionamiento de tu servidor al interactuar con un cliente HTTP (navegador web) utilizando los ejemplos de funcionamiento descritos en el apartado 2.

En las capturas de ejemplo debe aparecer tanto la consola de servidor como las del navegador web (cliente) con el que se hagan las peticiones al servidor, donde se pueda apreciar la barra de navegación con la URL solicitada y el resultado obtenido en el navegador de manera similar a como se muestra en el apartado 2.

El programa consiste en crear un servidor web básico el cual atendería a los clientes de forma secuencial. Este servidor atenderá sólo a las peticiones cuya ruta comience por la cadena “/saludar”, a esto se le denomina “contexto”. En el caso de que la petición no se ajuste a dicho contexto el servidor devolverá automáticamente el código de error HTTP 404. A continuación muestro una serie de ejemplos.

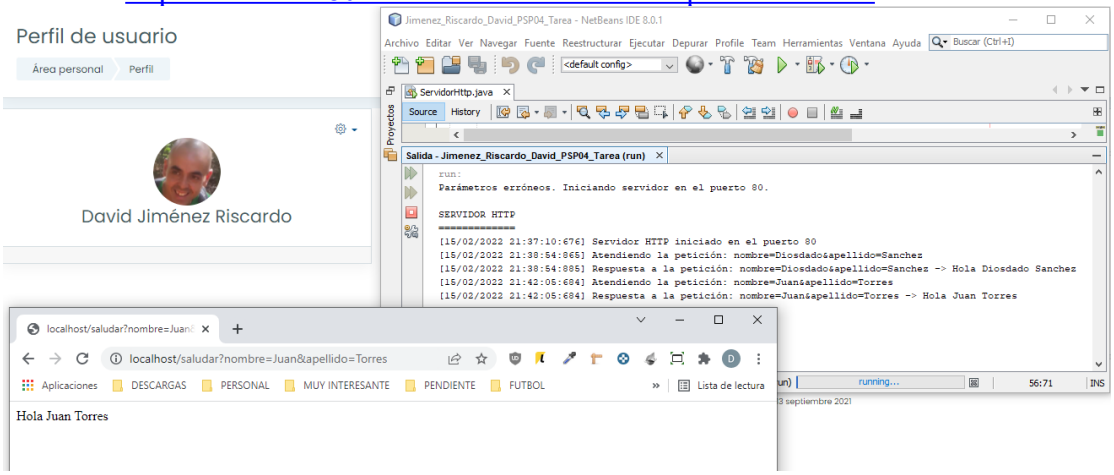
Ejemplo 1.

Petición: http://localhost:80/saludar?nombre=Diosdado&apellido=Sanchez



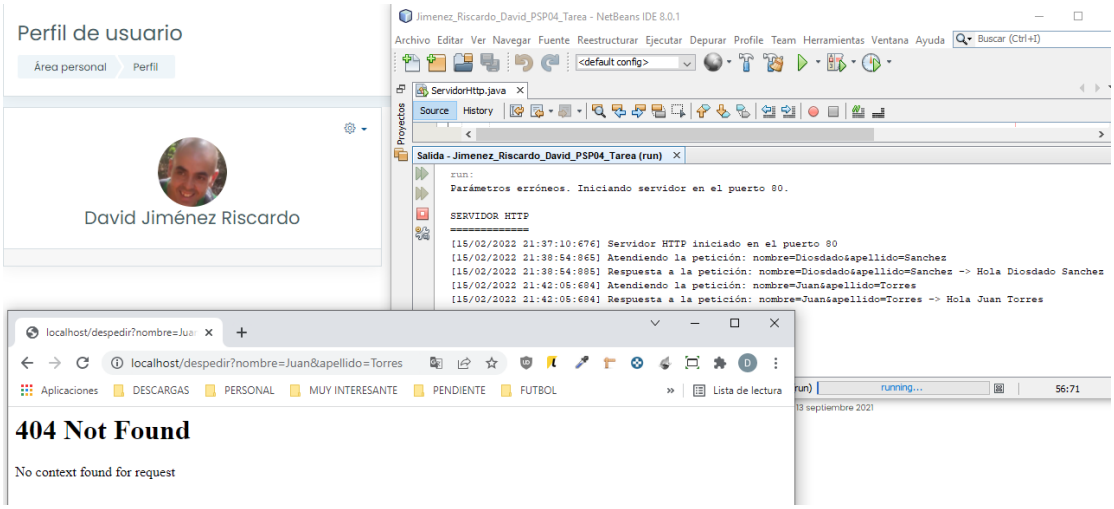
Ejemplo 2.

Petición: http://localhost:80/saludar?nombre=Juan&apellido=Torres



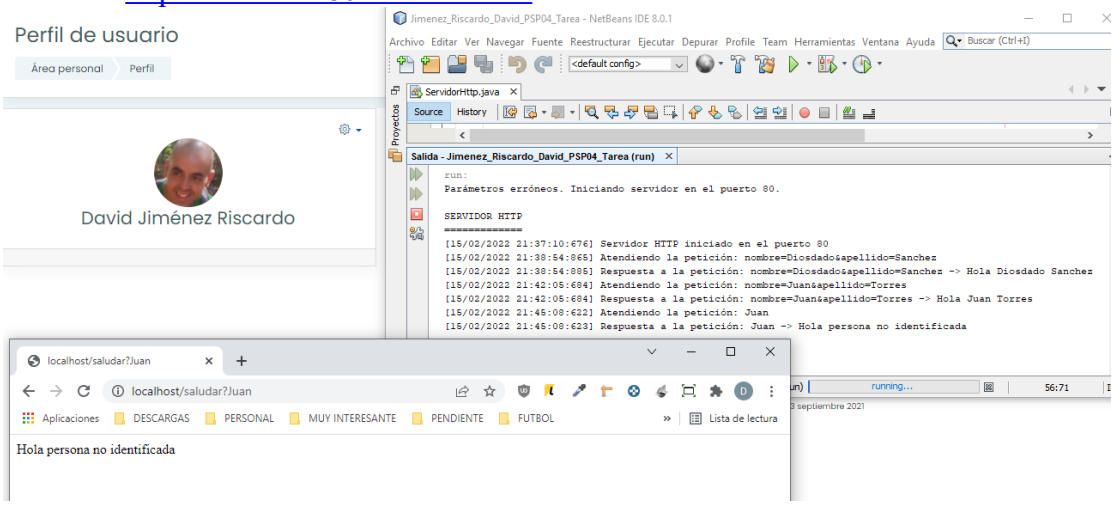
Ejemplo 3.

Petición: http://localhost:80/despedir?nombre=Juan&apellido=Torres



Ejemplo 4.

Petición: http://localhost:80/saludar?Juan



4.3 Ficha 3

Incluir breve explicación del funcionamiento del programa y capturas del funcionamiento de tu servidor al interactuar con un cliente HTTP (navegador web) utilizando los ejemplos de funcionamiento descritos en el apartado 3.
En las capturas debe aparecer la consola de servidor y varios navegadores donde se realicen:

1. un par de peticiones de "/primo" que tarde bastante en realizarse (con números primos relativamente altos, por ejemplo: 21341 y 10333)
2. tres peticiones "rápidas": otras dos peticiones de "/primo" que sean inmediatas (con número no primo: 100, y con una petición no válida), así como una petición de "/saludar", que también será inmediata. A estas solicitudes se les va a contestar antes, pues el servidor es multihilo y esas respuestas serán atendidas mientras las anteriores, más lentas, también están siendo procesadas.

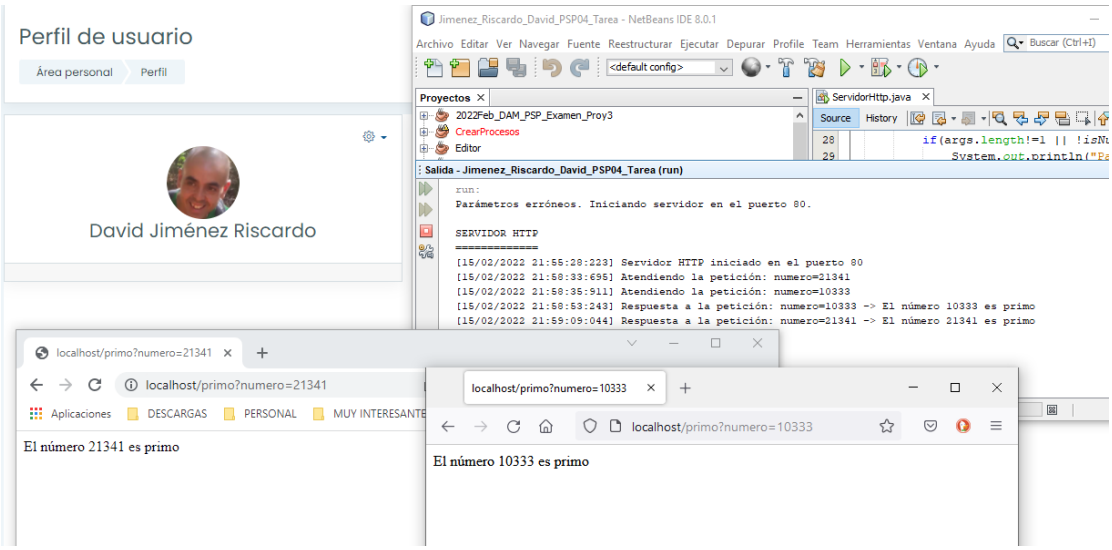
En la captura debe poder apreciarse que las dos primeras peticiones estarían aún resolviéndose mientras que las otras tres, posteriores, ya deberían haber sido resueltas.

En la captura de la consola del servidor debería apreciarse algo similar a lo que se observa en el ejemplo de funcionamiento del apartado 3.

Nuestro servidor web será capaz de atender a diferentes clientes de forma concurrente. A diferencia del anterior estará preparado para atender peticiones de dos contextos, "/saludar" y "/primo". Incluyo una serie de ejemplos.

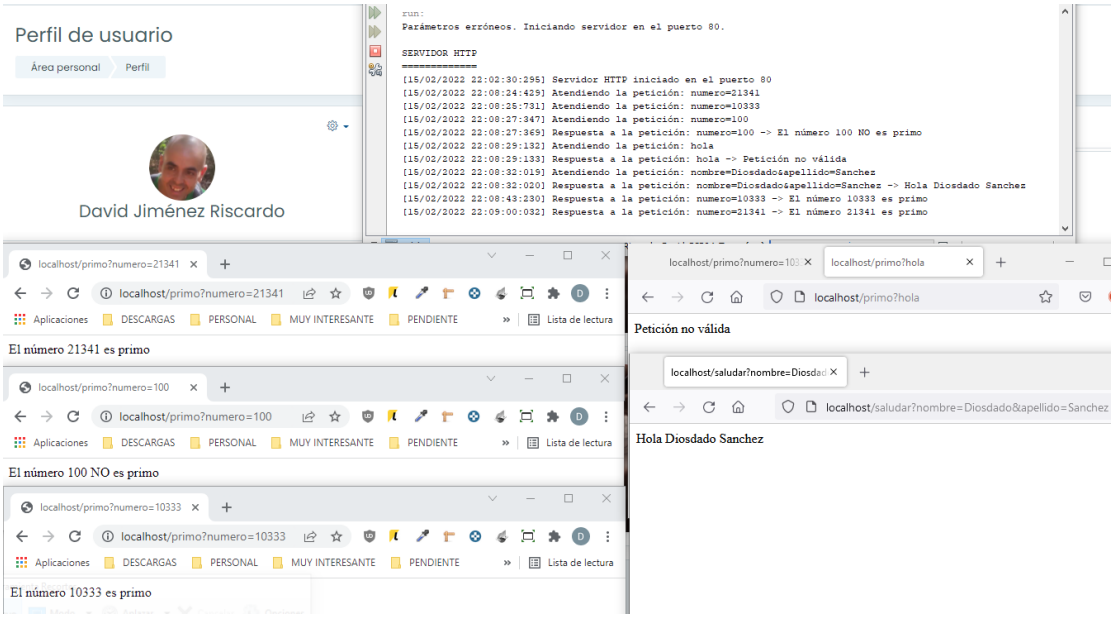
Ejemplo 1.

Peticiones: http://localhost:80/primo?numero=21341 y http://localhost:80/primo?numero=10333



Ejemplo 2.

Peticiones: http://localhost:80/primo?numero=21341, http://localhost:80/primo?numero=10333, http://localhost:80/primo?numero=100, http://localhost:80/primo?hola, http://localhost:80/saludar?nombre=Diosdado&apellido=Sanchez



Observamos en la consola que las peticiones que más procesamiento necesitan serán las últimas en resolverse.

5 Evaluación de la tarea

Cristerios de evaluación implicados

a) Se han analizado librerías que permitan implementar protocolos estándar de comunicación en red.

b) Se han programado clientes de protocolos estándar de comunicaciones y verificado su funcionamiento.

c) Se han desarrollado y probado servicios de comunicación en red.

d) Se han analizado los requerimientos necesarios para crear servicios capaces de gestionar varios clientes concurrentes.

e) Se han incorporado mecanismos para posibilitar la comunicación simultánea de varios clientes con el servicio.

f) Se ha verificado la disponibilidad del servicio.

g) Se han depurado y documentado las aplicaciones desarrolladas.

Nuestros programas no pueden abortar porque se produzca algún error inesperado que haga que salte alguna excepción no controlada que no capturemos y finalmente sea capturada por la máquina virtual de Java haciendo que el programa termine de manera abrupta.

¡ESO NO PUEDE SUCEDER!

Debes ser especialmente cuidadoso con:

- El control de posibles **errores con los parámetros y valores de entrada**: no superar los límites del array `args`, controlar si los parámetros cumplen los criterios especificados (por ejemplo si deben ser números: excepciones del tipo `NumberFormatException`, `InputMismatchException`), etc.
- Controlar si **los puertos que vamos a usar ya están siendo utilizados** (por ejemplo excepciones de tipo `BindException`).
- **Errores de E/S** en la gestión de archivos, sockets, flujos, etc.
- En general cualquier error que nosotros intuyamos que pueda producirse durante la ejecución de nuestro programa.

El hecho de que nuestros programas presenten este tipo de fragilidades y se "rompan" con esa facilidad eclipsa el buen hacer que hayamos podido desarrollar durante la implementación de nuestro código. Si durante la presentación de una aplicación, ésta falla nada más empezar vamos a producir una muy mala impresión a nuestros clientes, jefes, alumnos, profesores, asistentes, etc. echando por tierra todo el esfuerzo que hayamos dedicado a nuestro trabajo. Hay que cuidar este tipo de detalles en el acabado de nuestro producto.

Este tipo de fallos será severamente penalizado.

En la siguiente tabla se muestran los criterios de corrección aplicables a cada ejercicio, y la puntuación máxima asignada al mismo.

Criterios de corrección	*Puntuación*
• Se recibe como parámetro la `URL` que se desea solicitar a un servidor. • Se establece la conexión con el servidor en el host y puertos adecuados mediante el uso de las clases `URL` y `URLConnection`. • Se obtiene la **respuesta del servidor** y se muestra en la salida. • Se obtienen y se muestran en la salida las **cabeceras del servidor**. • En caso de tratarse de un documento HTML, es **descargado y almacenado localmente en un archivo**. • Se **finalizan adecuadamente todas las conexiones y se cierran apropiadamente todos los recursos** abiertos. • **Explicaciones y capturas** de funcionamiento en la **ficha de la tarea**. • No contiene elementos extraños, redundantes o sin sentido. • No se "rompe" con facilidad.	Hasta 3 puntos

• Es **configurable** desde **línea de órdenes** (**puerto**) • Se establecen todas las acciones necesarias para que el servidor esté a la escucha por el puerto adecuado y listo para recibir una conexión de cliente mediante el **uso de un objeto de la clase** `HttpServer`. • Se implementa un **objeto `HttpHandler` manejador del contexto "*/saludar*"** que cumple con los requisitos del enunciado. • Se muestran por consola los **mensajes informativos** indicados en el enunciado. • Finaliza adecuadamente todas las conexiones y cierra apropiadamente todos los recursos abiertos. • Explicaciones y capturas de funcionamiento en la ficha de la tarea. • No contiene elementos extraños, redundantes o sin sentido. • No se "rompe" con facilidad.	Hasta 3 puntos
• Es **configurable** desde **línea de órdenes** (**puerto**) • Se establecen todas las acciones necesarias para que el servidor esté a la escucha por el puerto adecuado y listo para recibir una conexión de cliente mediante el uso de un **objeto de la clase HttpServer** que es capaz de administrar un **pool de threads** para poder atender simultáneamente a varias peticiones. • Se implementa un objeto **`HttpHandler` manejador del contexto "*/primo*"** que cumple con los requisitos del enunciado. • Se muestran por consola los **mensajes informativos** indicados en el enunciado. • Finaliza adecuadamente todas las conexiones y cierra apropiadamente todos los recursos abiertos. • Explicaciones y capturas de funcionamiento en la ficha de la tarea. • No contiene elementos extraños, redundantes o sin sentido. • No se "rompe" con facilidad.	Hasta 4 puntos