El uso de Java HttpClient comienza poco a poco a ser más habitual ya que se trata de una librería de Java que viene incluida en la versión 9 y que nos permite de una forma muy natural realizar peticiones REST. Vamos a apoyarnos en un servicio REST construido con Spring Boot que únicamente tiene un método que nos devuelve un mensaje.

```
package com.arquitecturajava.rest;

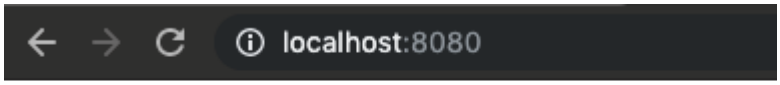
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HolaController {

  @GetMapping
  public String mensaje() {

    try {
      Thread.sleep(3000);
    } catch (InterruptedException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    return "hola desde spring";
  }
}
```

Si lanzamos el servicio y nos conectamos a el vía un navegador podremos acceder a su información de forma directa la cual nos tardará unos 3 segundos en aparecer.



hola desde spring

Java HttpClient

Es momento de acceder a esa información utilizando el nuevo cliente Http de Java 9 (HttpClient) . Este nos permitirá un acceso de forma muy directa a los datos e imprimirlos en una consola. Vamos a ver el código:

```
package com.arquitecturajava;

import java.io.IOException;
import java.net.URI;

import jdk.incubator.http.HttpClient;
import jdk.incubator.http.HttpRequest;
import jdk.incubator.http.HttpResponse;
import jdk.incubator.http.HttpResponse.BodyHandler;

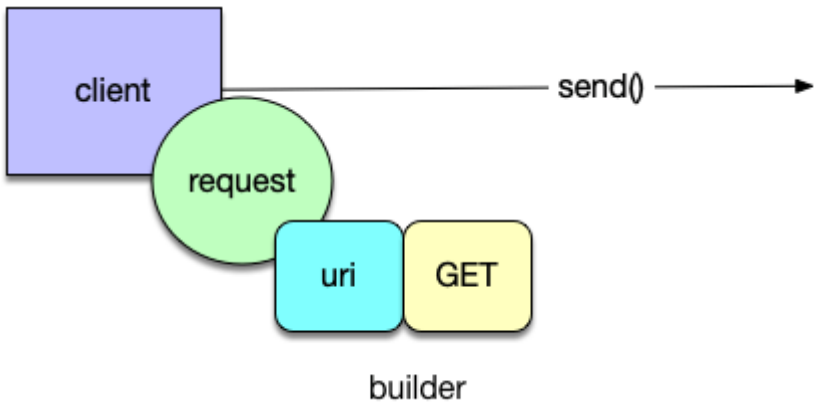
public class Principal {

  public static void main(String[] args) {
     HttpClient client = HttpClient.newHttpClient();
        HttpRequest request = HttpRequest.newBuilder()
              .uri(URI.create("http://localhost:8080"))
              .GET()
              .build();
        try {
        HttpResponse<String> respuesta=client.send(request,
BodyHandler.asString());
        System.out.println(respuesta.body());
```

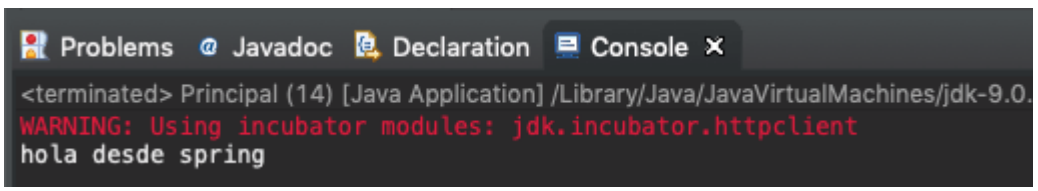
```
        } catch (IOException | InterruptedException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
        }
    }

}
```

De esta manera utilizando un Java HttpClient podemos acceder a la url sin tener que incluir librerías de terceros. En este caso la clase hace uso de un builder que solicita primero la uri a la que deseamos acceder , define el verbo que vamos a utilizar para el acceso. En este caso será GET y con ello se construye un objeto de tipo HttpRequest.



Este es el objeto que vía HttpClient es procesado y se realiza una petición GET al servidor que devolverá unos datos y los imprimimos por la consola.



Acabamos de hacer uso de la nueva clase de Java 9 . Nos puede parecer un ejemplo realmente sencillo pero esta clase admite muchas opciones orientadas a la programación asíncrona que encajan de una forma muy natural y sencilla a la hora de resolver situaciones complejas. Vamos a ver un ejemplo en el cual realizamos dos peticiones a la misma URL ,

cada una de ellas tardará 3 segundos . Pero nosotros utilizando Java HttpClient lanzaremos las peticiones de forma simultanea y cuando lleguen los resultados las combinaremos utilizando CompletableFuture.

```
package com.arquitecturajava;

import java.net.URI;
import java.util.concurrent.CompletableFuture;
import java.util.stream.Collectors;
import java.util.stream.Stream;

import jdk.incubator.http.HttpClient;
import jdk.incubator.http.HttpRequest;
import jdk.incubator.http.HttpResponse;
import jdk.incubator.http.HttpResponse.BodyHandler;

public class Principal2 {

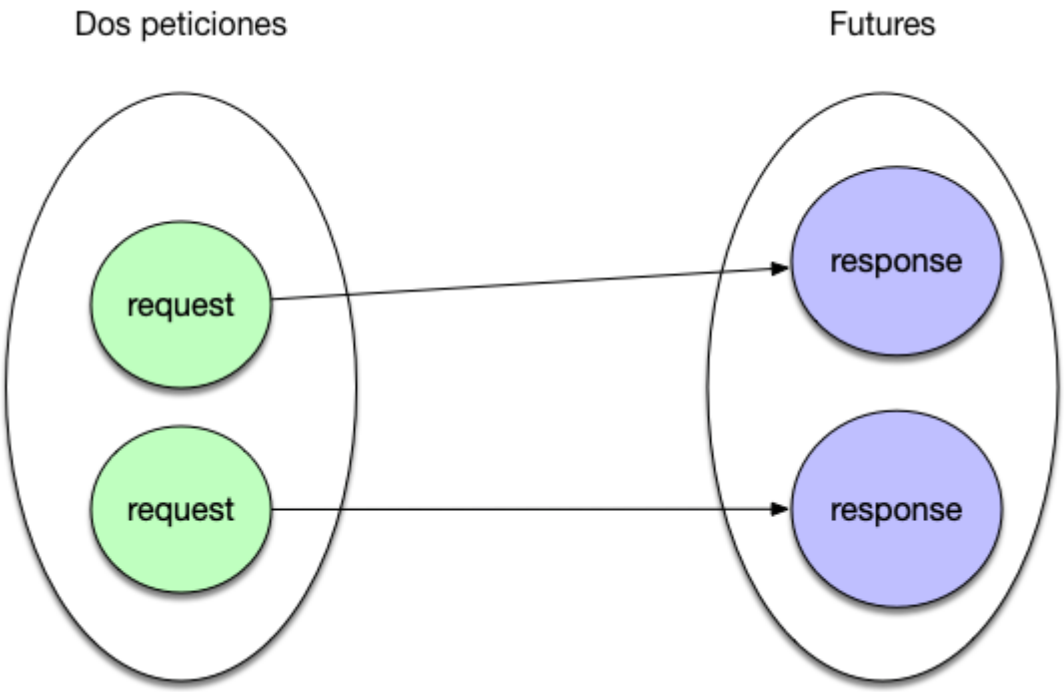
  public static void main(String[] args) {
     HttpClient client = HttpClient.newHttpClient();
        HttpRequest request = HttpRequest.newBuilder()
              .uri(URI.create("http://localhost:8080"))
              .GET()
              .build();
        HttpClient client2 = HttpClient.newHttpClient();

        CompletableFuture<String> respuesta=client
                        .sendAsync(request, BodyHandler.asString())
                        .thenApply(HttpResponse::body);
        CompletableFuture<String> respuesta2=client2
                        .sendAsync(request, BodyHandler.asString())
```

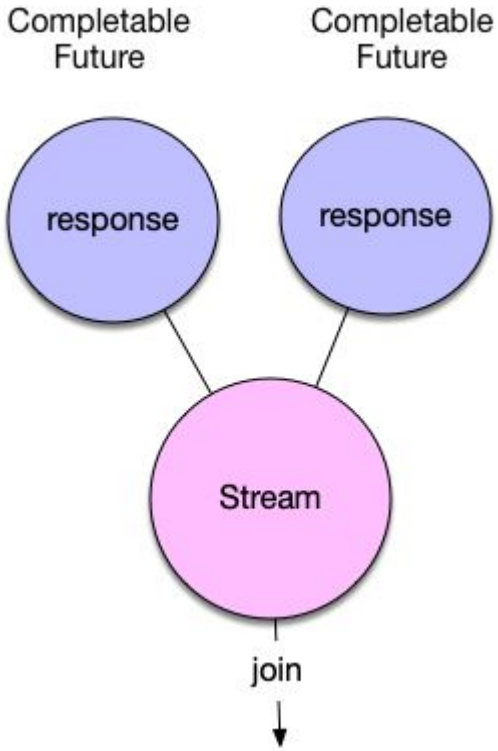
```
                        .thenApply(HttpResponse::body);
    String resultado = Stream.of(respuesta,
respuesta2).map(CompletableFuture::join)
        .collect(Collectors.joining(" "));
    System.out.println(resultado);
  }

}
```

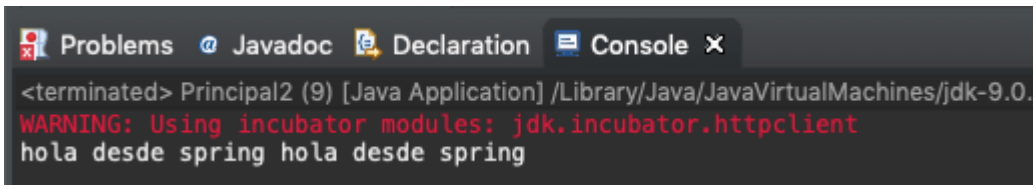
En este caso lo que tenemos son dos clientes , cada uno de ellos hace una petición HTTP de forma simultanea y cuando los resultados llegan al cliente los combinamos .



Esto el final se parece mucho al uso clásico del api de promesas de JavaScript pero pasado a Java



De esta forma y al cabo de 3 segundos obtendremos un resultado que combina ambas peticiones.



Otros artículos relacionados

1. Java Stream Filter y Predicates
2. JavaScript Promise y la programación asíncrona
3. Java Parallel Stream y rendimiento
4. java 9