HTTP-клиент в Java 9

Общей четкой одинаковой конвы для написания кода с ипользованием HTTP-клиента нет. Каждый конкретный случай будет писаться посвоему. Т.е. нужно смотреть написание кода для каждой конкретной задачи, везде будут свои нюансы.

**1. Вступление**

До недавнего времени Java предоставляла только API-интерфейс *HttpURLConnection*, который является низкоуровневым, малофункциональным и неудобным для пользователя.

Поэтому обычно использовались некоторые широко используемые сторонние библиотеки, такие как [Apache HttpClient](https://hc.apache.org/httpcomponents-client-ga/), [Jetty](https://www.eclipse.org/jetty/documentation/current/http-client-api.html)и ссылка Spring:/rest-template[RestTemplate].

**2. Начальная настройка**

* Модуль HTTP-клиента упакован как инкубаторный модуль \*\* в JDK 9 и поддерживает [HTTP/2](https://en.wikipedia.org/wiki/HTTP/2)с обратной совместимостью с HTTP/1.1.

Среди преимуществ HTTP 2 можно выделить следующие:

* **Мультиплексирование**HTTP  2 может отправлять несколько запросов данных параллельно, используя одно TCP соединение. В HTTP 1.1 мы не можем использовать более 6 открытых соединений одновременно.
* **Замена текста на Binary**
* **Server Push**При работе с HTTP 1.1 страницы пересылаются на клиент (браузеру) и он должен парсить HTML. Во время этого процесса, клиент может запрашивать дополнительные данные от сервера. В HTTP 2 сервер может отправлять эти данные без запроса от клиента

Чтобы использовать его, нам нужно определить наш модуль с помощью файла *module-info.java*, в котором также указать модуль, необходимый для запуска нашего приложения:

**module** myModule {

**requires** jdk.incubator.httpclient;

}

**3. Обзор API HTTP-клиента**

В отличие от *HttpURLConnection,*HTTP Client предоставляет механизмы синхронного и асинхронного запроса

API состоит из 3 основных классов:

* ***HttpRequest*****-**представляет запрос, который будет отправлен через *HttpClient , (*используется для создания HTTP запроса от клиента к серверу)
* ***HttpClient***- ведет себя как контейнер для информации о конфигурации общей для нескольких запросов (отвечает за обработку и отправку HTTP запросов)
* ***HttpResponse***- представляет результат вызова *HttpRequest (*содержит HTTP ответ от сервера)

**4. *HttpRequest***

*HttpRequest*- это объект, который представляет запрос, который мы хотим отправить. Новые экземпляры создаются с помощью *HttpRequest.Builder.* Мы можем получить это, вызвав *HttpRequest.newBuilder ()*.

Класс *Builder*предоставляет набор методов, которые мы можем использовать для настройки нашего запроса. Мы рассмотрим самые важные из них.

**4.1. Настройка *URI***

Первое, что мы должны сделать при создании запроса, это предоставить URL.

Мы можем сделать это двумя способами - используя конструктор для *Builder*с параметром *URI*или вызывая метод *uri (URI)*для экземпляра *Builder*:

HttpRequest.newBuilder(**new** URI("https://postman-echo.com/get"))

HttpRequest.newBuilder().uri(**new** URI("https://postman-echo.com/get"))

Последнее, что нам нужно настроить для создания базового запроса, - это метод HTTP.

**4.2. Указание метода HTTP**

Мы можем определить HTTP-метод, который будет использовать наш запрос, вызвав один из методов из *Builder*:

* *GET()*
* *POST (BodyProcessor body)*
* *PUT (BodyProcessor body)*
* *DELETE (тело BodyProcessor)*

Мы расскажем *BodyProcessor*подробно, позже. Теперь давайте просто создадим **очень простой пример запроса GET**:

HttpRequest request = HttpRequest.newBuilder()

.uri(**new** URI("https://postman-echo.com/get"))

.GET()

.build();

Этот запрос имеет все параметры, требуемые *HttpClient*. Однако иногда нам нужно добавить дополнительные параметры в наш запрос; Вот некоторые важные из них:

* версия протокола HTTP
* заголовки
* тайм-аут

**4.3. Настройка версии протокола HTTP (*version*)**

API полностью использует протокол HTTP/2 и использует его по умолчанию, но мы можем определить, какую версию протокола мы хотим использовать.

HttpRequest request = HttpRequest.newBuilder()

.uri(**new** URI("https://postman-echo.com/get"))

.version(HttpClient.Version.HTTP\_\_2)

.GET()

.build();

Здесь важно упомянуть, что клиент будет использовать, например, HTTP/1.1, если HTTP/2 не поддерживается. \*\*

**4.4. Настройка заголовков (*headers*)**

В случае, если мы хотим добавить дополнительные заголовки к нашему запросу, мы можем использовать предоставленные методы компоновщика.

Мы можем сделать это одним из двух способов:

* передача всех заголовков в виде пар ключ-значение в метод *headers ()*или
* используя метод *header ()*для единственного заголовка значения ключа:

HttpRequest request = HttpRequest.newBuilder()

.uri(**new** URI("https://postman-echo.com/get"))

.headers("key1", "value1", "key2", "value2")

.GET()

.build();

HttpRequest request2 = HttpRequest.newBuilder()

.uri(**new** URI("https://postman-echo.com/get"))

.header("key1", "value1")

.header("key2", "value2")

.GET()

.build();

Последний полезный метод, который мы можем использовать для настройки нашего запроса, это *timeout ()*.

**4.5. Установка таймаута (*timeout*)**

Давайте теперь определим количество времени, которое мы хотим ждать ответа.

Если установленное время истекает, генерируется *HttpTimeoutException*;

Время ожидания по умолчанию установлено в бесконечность.

Время ожидания можно установить с помощью объекта *Duration*, вызвав метод *timeout ()*для экземпляра компоновщика:

HttpRequest request = HttpRequest.newBuilder()

.uri(**new** URI("https://postman-echo.com/get"))

.timeout(Duration.of(10, SECONDS))

.GET()

.build();

**5. Настройка тела запроса**

Мы можем добавить тело к запросу, используя методы построителя запросов:

*POST (тело BodyProcessor)*, *PUT (тело BodyProcessor)*и *DELETE (тело BodyProcessor) .*

Новый API предоставляет ряд готовых реализаций *BodyProcessor*, которые упрощают передачу тела запроса:

* *StringProcessor*(читает тело из *String*, созданного с*HttpRequest.BodyProcessor.fromString*)
* *InputStreamProcessor*(читает тело из *InputStream*, созданного с *HttpRequest.BodyProcessor.fromInputStream*)
* *ByteArrayProcessor*(читает тело из байтового массива, созданного с помощью *HttpRequest.BodyProcessor.fromByteArray*)
* *FileProcessor*(читает тело из файла по заданному пути, созданного с *HttpRequest.BodyProcessor.fromFile*)

Если нам не нужно тело, мы можем просто передать *HttpRequest.noBody ()*:

HttpRequest request = HttpRequest.newBuilder()

.uri(**new** URI("https://postman-echo.com/post"))

.POST(HttpRequest.noBody())

.build();

**5.1. *StringBodyProcessor***

Задать тело запроса в любой реализации *BodyProcessor*очень просто и интуитивно понятно.

Например, если мы хотим передать простой *String*в качестве тела, мы можем использовать *StringBodyProcessor*.

Как мы уже упоминали, этот объект может быть создан с помощью фабричного метода *fromString ()*; он принимает в качестве аргумента только объект *String*и создает из него тело:

HttpRequest request = HttpRequest.newBuilder()

.uri(**new** URI("https://postman-echo.com/post"))

.headers("Content-Type", "text/plain;charset=UTF-8")

.POST(HttpRequest.BodyProcessor

.fromString("Sample request body"))

.build();

**5.2. *InputStreamBodyProcessor***

Чтобы сделать это, *InputStream*должен быть передан как *Supplier*(чтобы сделать его создание ленивым), поэтому он немного отличается от описанного выше *StringBodyProcessor*.

Тем не менее, это также довольно просто:

**byte**[]sampleData = "Sample request body".getBytes();

HttpRequest request = HttpRequest.newBuilder()

.uri(**new** URI("https://postman-echo.com/post"))

.headers("Content-Type", "text/plain;charset=UTF-8")

.POST(HttpRequest.BodyProcessor

.fromInputStream(() -> **new** ByteArrayInputStream(sampleData)))

.build();

Обратите внимание, как мы использовали здесь простой *ByteArrayInputStream*; это, конечно, может быть любая реализация *InputStream*.

**5.3. *ByteArrayProcessor***

Мы также можем использовать *ByteArrayProcessor*и передать массив байтов в качестве параметра:

**byte**[]sampleData = "Sample request body".getBytes();

HttpRequest request = HttpRequest.newBuilder()

.uri(**new** URI("https://postman-echo.com/post"))

.headers("Content-Type", "text/plain;charset=UTF-8")

.POST(HttpRequest.BodyProcessor

.fromByteArray(sampleData))

.build();

**5.4. *FileProcessor***

Для работы с файлом мы можем использовать предоставленный *FileProcessor*; его фабричный метод берет путь к файлу в качестве параметра и создает тело из содержимого:

HttpRequest request = HttpRequest.newBuilder()

.uri(**new** URI("https://postman-echo.com/post"))

.headers("Content-Type", "text/plain;charset=UTF-8")

.POST(HttpRequest.BodyProcessor

.fromFile(Paths.get("src/test/resources/sample.txt")))

.build();

Мы рассмотрели, как создать *HttpRequest*и как установить в нем дополнительные параметры

Теперь пришло время глубже взглянуть на класс *HttpClient*, который отвечает за отправку запросов и получение ответов.

**6. *HttpClient***

Он предоставляет множество методов, которые мы можем использовать **для обработки** нашего **запроса/ответа**.

Давайте рассмотрим некоторые из них здесь.

**6.1. Настройка прокси**

Мы можем определить прокси для подключения. Просто вызовите метод *proxy ()*для экземпляра *Builder*:

HttpResponse<String> response = HttpClient.newBuilder()

.proxy(ProxySelector.getDefault())

.build()

.send(**request**, HttpResponse.BodyHandler.ofString()); **//в реальной жизни вместо “request”**

**будет находится запрос и его тело (из пунктов 4 и 5) +**

**используется ofString() BodyHandler для преобразования байтов тела ответа в String**

В нашем примере мы использовали системный прокси по умолчанию.

ProxySelector может быть настроен через метод клиента Builder::proxy. ProxySelectorAPI возвращает определенный прокси-сервер для данного URI. Во многих случаях достаточно одного статического прокси.  Для создания такого селектора может быть использован статический метод ProxySelector::of.

public CompletableFuture<String> get(String uri) {

HttpClient client = HttpClient.newBuilder()

.proxy(ProxySelector.of(new InetSocketAddress("www-proxy.com", 8080)))

.build();

HttpRequest request = HttpRequest.newBuilder()

.uri(URI.create(uri))

.build();

return client.sendAsync(request, BodyHandlers.ofString())

.thenApply(HttpResponse::body);

**6.2. Настройка политики перенаправления**

Иногда страница, к которой мы хотим получить доступ, перемещалась по другому адресу.

В этом случае мы получим код состояния HTTP 3xx, обычно с информацией о новом URI.

***HttpClient*может автоматически перенаправить запрос на новый URI, если мы установим соответствующую политику перенаправления.**

Мы можем сделать это с помощью метода *followRedirects ()*в *Builder*:

HttpResponse<String> response = HttpClient.newBuilder()

.followRedirects(HttpClient.Redirect.ALWAYS)

.build()

.send(request, HttpResponse.BodyHandler.ofString());

Все политики определены и описаны в enum *HttpClient.Redirect*.

**6.3. Настройка *Authenticator*для подключения**

*Authenticator*- это объект, который согласовывает учетные данные (аутентификацию HTTP) для соединения

Он предоставляет различные схемы аутентификации (например, базовую или дайджест-аутентификацию). В большинстве случаев для аутентификации требуется имя пользователя и пароль для подключения к серверу.

Мы можем использовать класс *PasswordAuthentication*, который является просто держателем этих значений:

HttpResponse<String> response = HttpClient.newBuilder()

.authenticator(**new** Authenticator() {

@Override

**protected** PasswordAuthentication **getPasswordAuthentication**() {

**return** **new** PasswordAuthentication(

"username",

"password".toCharArray());

}

}).build()

.send(request, HttpResponse.BodyHandler.ofString());

В приведенном выше примере мы передали значения имени пользователя и пароля в виде открытого текста; конечно в реальной программе это должно быть иначе.

Обратите внимание, что не каждый запрос должен использовать одинаковые имя пользователя и пароль.

Класс *Authenticator*предоставляет несколько методов *getXXX*(например, *getRequestingSite ()*), которые можно использовать для определения того, какие значения следует предоставить.

Теперь мы собираемся изучить одну из самых полезных функций нового *HttpClient*- асинхронные вызовы на сервер.

**6.4. Отправить запросы - Sync vs. Async**

Новый HttpClient предоставляет две возможности для отправки запроса на сервер:

* ***send (…​)*- синхронно**(блокируется до получения ответа)
* ***sendAsync (…​)*- асинхронно**(не ждет ответа,без блокировки)

Используемый ранее метод *send (…​)*естественно ожидал ответа:

HttpResponse<String> response = HttpClient.newBuilder()

.build()

.**send**(request, HttpResponse.BodyHandler.ofString());

Этот вызов возвращает объект *HttpResponse*, и мы уверены, что следующая инструкция из нашего потока приложения будет выполнена только тогда, когда ответ уже здесь.

Однако у него много недостатков, особенно когда мы обрабатываем большие объемы данных.

Итак, теперь мы можем использовать метод *sendAsync (…​)*, который возвращает *CompletableFeature <HttpResponse>*- **для асинхронной обработки запроса**:

CompletableFuture<HttpResponse<String>> response = HttpClient.newBuilder()

.build()

.sendAsync(request, HttpResponse.BodyHandler.ofString());

Новый API также может работать с несколькими ответами и передавать тела запросов и ответов (*тут будет дикая смесь из CompletableFuture, Stream.API, HttpClient*):

List<URI> targets = Arrays.asList(

**new** URI("https://postman-echo.com/get?foo1=bar1"),

**new** URI("https://postman-echo.com/get?foo2=bar2"));

HttpClient client = HttpClient.newHttpClient();

List<CompletableFuture<String>> futures = targets.stream()

.map(target -> client //тут работает метод Stream.API

.**sendAsync** //тут внутри Stream.API для “client” (который класса “HttpClient”) начинает

работать метод из HttpClient “sendAsync()”

**(**HttpRequest.newBuilder(target).GET().build(), HttpResponse.BodyHandler.ofString()**)**

.thenApply(response -> response.body())) //тут опять начинают работать методы Stream.Api, а

body() это метод для “Response” из главы 7

.collect(Collectors.toList());

**6.5. Установка *Executor*для асинхронных вызовов**

Мы также можем определить *Executor*, который предоставляет потоки для асинхронных вызовов.

Таким образом, мы можем, например, ограничить количество потоков, используемых для обработки запросов:

ExecutorService executorService = Executors.newFixedThreadPool(2);

CompletableFuture<HttpResponse<String>> response1 = HttpClient.newBuilder()

.executor(executorService)

.build()

.sendAsync(request, HttpResponse.BodyHandler.ofString());

CompletableFuture<HttpResponse<String>> response2 = HttpClient.newBuilder()

.executor(executorService)

.build()

.sendAsync(request, HttpResponse.BodyHandler.ofString());

По умолчанию *HttpClient*использует executor *java.util.concurrent.Executors.newCachedThreadPool ()*.

**6.6. Определение *CookieManager***

С новым API и компоновщиком легко установить *CookieManager*для нашего соединения. Мы можем использовать метод построителя *cookieManager (CookieManager cookieManager)*, чтобы определить специфичный для клиента *CookieManager*.

Давайте, например, определим *CookieManager*, который вообще не позволяет принимать куки:

HttpClient.newBuilder()

.cookieManager(**new** CookieManager(**null**, CookiePolicy.ACCEPT\_\_NONE))

.build();

Если наш *CookieManager*позволяет сохранять файлы cookie, мы можем получить к ним доступ, проверив *CookieManager*из нашего *HttpClient*:

httpClient.cookieManager().get().getCookieStore()

Теперь давайте сосредоточимся на последнем классе API Http - *HttpResponse*.

**7. *HttpResponse*Object**

Класс *HttpResponse*представляет ответ от сервера. Он предоставляет ряд полезных методов:

* *body ()*- возвращает тело ответа/ получаем ответ на запрос (тип возвращаемого значения зависит от параметр *BodyHandler* ответа, переданный методу *send ()*)

Остальные методы *HttpResponse* это получение информации о самом запросе или способе работы с запросом.

* *statusCode ()*- возвращает код состояния (тип *int*) для ответа (Класс *HttpURLConnection* содержит возможные значения). Также есть методы *uri ()*, *headers ()*, *trailers ()*и *version ()*.

**7.1. *URI*объекта ответа**

Метод *uri ()*объекта ответа возвращает *URI*, от которого мы получили ответ.

Иногда он может отличаться от *URI*в объекте запроса, поскольку может произойти перенаправление

assertThat(request.uri()

.toString(), equalTo("http://stackoverflow.com"));

assertThat(response.uri()

.toString(), equalTo("https://stackoverflow.com/"));

**7.2. Заголовки из ответа**

Мы можем получить заголовки из ответа, вызвав метод *headers ()*для объекта ответа:

HttpResponse<String> response = HttpClient.newHttpClient()

.send(request, HttpResponse.BodyHandler.ofString());

HttpHeaders responseHeaders = response.headers();

Возвращает объект *HttpHeaders*в качестве возвращаемого типа. Это новый тип, определенный в пакете *jdk.incubator.http*, который представляет представление заголовков HTTP только для чтения.

У него есть несколько полезных методов, которые упрощают поиск значения заголовков.

**7.3. Получить трейлеры от ответа**

HTTP-ответ может содержать дополнительные заголовки, которые включаются после содержимого ответа. Эти заголовки называются заголовками трейлера.

Мы можем получить их, вызвав метод *trailers ()*для *HttpResponse:*

HttpResponse<String> response = HttpClient.newHttpClient()

.send(request, HttpResponse.BodyHandler.ofString());

CompletableFuture<HttpHeaders> trailers = response.trailers();

Обратите внимание, что метод *trailers ()*возвращает объект *CompletableFuture*.

**7.4. Версия Ответа**

Метод *version ()*определяет, какая версия протокола HTTP использовалась для связи с сервером.

* Помните, что даже если мы определим, что хотим использовать HTTP/2, сервер может ответить через HTTP/1.1. \*\*

Версия, в которой ответил сервер, указана в ответе:

HttpRequest request = HttpRequest.newBuilder()

.uri(**new** URI("https://postman-echo.com/get"))

.version(HttpClient.Version.HTTP\_\_2)

.GET()

.build();

HttpResponse<String> response = HttpClient.newHttpClient()

.send(request, HttpResponse.BodyHandler.ofString());

assertThat(response.version(), equalTo(HttpClient.Version.HTTP\_\_1\_\_1));

## 8. HttpResponse.BodyHandlers

## С помощью методов класса [BodyHandler](https://docs.oracle.com/en/java/javase/11/docs/api/java.net.http/java/net/http/HttpResponse.BodyHandler.html" \o "интерфейс в java.net.http) *(их около 20)* мы можем получать ответы на запрос разными способами, например в виде строки или записи тела ответа в файл.

## Некоторые из них накапливают байты ответа в памяти до тех пор, пока он не будет полностью получен, после чего он преобразуется, например ofString, и ofByteArray.  Другие передают данные ответа по мере их поступления: ofFile, ofByteArrayConsumer и ofInputStream.

В следующем примере используется один из [предопределенных обработчиков тела,](https://docs.oracle.com/en/java/javase/11/docs/api/java.net.http/java/net/http/HttpResponse.BodyHandlers.html) который всегда обрабатывают тело ответа одним и тем же способом (передает тело ответа в файл).

HttpRequest request = HttpRequest.newBuilder()

.uri(URI.create("http://www.foo.com/"))

.build();

client.sendAsync(request, BodyHandlers.ofFile(Paths.get("/tmp/f")))

.thenApply(HttpResponse::body) //так же как и в примере в 6.4. Видно именно эта строка вызывает тело ответа, чтобы потом обработать его так как написано в строке выше в методе BodyHandlers.

.thenAccept(System.out::println); //тут thenAccept() т.к. никакой результат не возвращается, а просто выполняется действие строки кода

**9. Заключение**

В этой статье мы рассмотрели *HttpClient*API Java 9, который обеспечивает большую гибкость и мощные функции. Как всегда, полный код можно найти на [over на GitHub](https://github.com/eugenp/tutorials/tree/master/core-java-9).

**Еще примеры:**

**1. JSON.** Использование библиотеки Джексона в сочетании  BodyHandlers::ofString:

1.1. преобразование ответа JSON в Map пары ключей / значений String.

public CompletableFuture<Map<String,String>> JSONBodyAsMap(URI uri) {

UncheckedObjectMapper objectMapper = new UncheckedObjectMapper();

HttpRequest request = HttpRequest.newBuilder(uri)

.header("Accept", "application/json")

.build();

return HttpClient.newHttpClient()

.sendAsync(request, BodyHandlers.ofString())

.thenApply(HttpResponse::body)

.thenApply(objectMapper::readValue);

}

class UncheckedObjectMapper extends com.fasterxml.jackson.databind.ObjectMapper {

/\*\* Parses the given JSON string into a Map. \*/

Map<String,String> readValue(String content) {

try {

return this.readValue(content, new TypeReference<>(){});

} catch (IOException ioe) {

throw new CompletionException(ioe);

}

}

1.2. преобразования Map пар ключей / значений String в JSON.

public CompletableFuture<Void> postJSON(URI uri,

Map<String,String> map)

throws IOException

{

ObjectMapper objectMapper = new ObjectMapper();

String requestBody = objectMapper

.writerWithDefaultPrettyPrinter()

.writeValueAsString(map);

HttpRequest request = HttpRequest.newBuilder(uri)

.header("Content-Type", "application/json")

.POST(BodyPublishers.ofString(requestBody))

.build();

return HttpClient.newHttpClient()

.sendAsync(request, BodyHandlers.ofString())

.thenApply(HttpResponse::statusCode)

.thenAccept(System.out::println);

}

### 2. Почта

public void post(String uri, String data) throws Exception {

HttpClient client = HttpClient.newBuilder().build();

HttpRequest request = HttpRequest.newBuilder()

.uri(URI.create(uri))

.POST(BodyPublishers.ofString(data))

.build();

HttpResponse<?> response = client.send(request, BodyHandlers.discarding());

System.out.println(response.statusCode());

}

В приведенном выше примере используется BodyPublisher.ofString для преобразования данных String в байты тела запроса.

Это BodyPublisher издатель реактивного потока, который публикует потоки тела запроса по требованию.

HttpRequest.Builder имеет ряд методов, позволяющих установить BodyPublisher: Builder::POST, Builder::PUT и Builder::method.

У HttpRequest.BodyPublishers класса есть несколько удобных статических фабричных методов, которые создают BodyPublisher для общих типов данных: ofString, ofByteArray, ofFile.

discarding BodyHandlerМожет быть использован для приема и отбросить тело ответа , когда он не представляет интереса.

Рассмотрим небольшой **живой** пример (*получение HTML кода страницы* google.com ):

Добавим в корневую директорию файл module-info.java:

module net.proselyte.java9 {

requires jdk.incubator.httpclient;

}

Это необходимо для импорта в наш текущий модуль jdk.incubator.httpclient.

package net.proselyte.java9;

import jdk.incubator.http.\*;

import java.io.IOException;

import java.net.URI;

import java.net.URISyntaxException;

public class Http2Demo {

public static void main(String[] args) throws URISyntaxException, IOException,

InterruptedException

{

HttpClient httpClient = HttpClient.newHttpClient();

System.out.println("httpClient version: " + httpClient.version());

HttpRequest httpRequest = HttpRequest.newBuilder().uri(new URI

"https://www.google.com/")).GET().build();

HttpResponse httpResponse = httpClient.send(httpRequest,

HttpResponse.BodyHandler.ofString());

System.out.println("httpResponse body: " + httpResponse.body());

System.out.println("httpResponse body: " + httpResponse.statusCode());

}

}

В результате работы программы мы получим, примерно, следующий результат:

WARNING: Using incubator modules: jdk.incubator.httpclient

httpClient version: HTTP\_2

**httpResponse body:** <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="uk"><head><meta content="text/html; charset=UTF-8" http-equiv="Content-Type"><meta content="/images/branding/googleg/1x/googleg\_standard\_color\_128dp.png" itemprop="image"><title>Google</title><script - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

(){var r=[];google.plm(r);})();(function(){var m=[]

;google.jsc && google.jsc.m(m);})();</script></div></body></html>

**httpResponse body:** 200

Кроме того, мы имеем возможность работать с асинхронными запросами:

package net.proselyte.java9;

import jdk.incubator.http.HttpClient;

import jdk.incubator.http.HttpRequest;

import jdk.incubator.http.HttpResponse;

import java.net.URI;

import java.net.URISyntaxException;

import java.util.concurrent.CompletableFuture;

import java.util.concurrent.ExecutionException;

public class Http2DemoAsync {

public static void main(String[] args) throws URISyntaxException,

InterruptedException, ExecutionException

{

HttpClient httpClient = HttpClient.newHttpClient();

System.out.println("httpClient version: " + httpClient.version());

HttpRequest httpRequest = HttpRequest.newBuilder().uri(new URI

"https://www.google.com/")).GET().build();

CompletableFuture<HttpResponse> httpResponse = httpClient.sendAsync(httpRequest,

HttpResponse.BodyHandler.ofString());

if (httpResponse.isDone())

{

System.out.println("httpResponse body: " + httpResponse.get().body());

System.out.println("httpResponse body: " +

httpResponse.get().statusCode());

}

else

{

System.out.println("Response is not completed...");

httpResponse.cancel(true); //прерываем задание

}

}

}

В результате выполнения данного примера, мы получим:

(*тут наверное одним потоком выполняется запрос, и одновременно паралельно другим потоком запрашивается результат выполнения и поэтому получаем ответ что невыполнено*)

WARNING: Using incubator modules: jdk.incubator.httpclient

httpClient version: HTTP\_2

Response is not completed...