Java HTTP Client

Azat Satklichov

azats@seznam.cz,
https://github.com/azatsatklichov/Java-Features/tree/master/src/main/java/features/in/java11/httpclient

Agenda

❑ Java HTTP Client Libraries

❑ Java HttpURLConnection

❑ Introducing HttpClient

❑ BodyHandler, Synch-Async Requests

❑ HttpClient Configuration

❑ BodyPublisher

❑ Headers & Cookies

❑ Security (Secure connections, HTTP Basic Auth)

❑ Advanced HttpClient Features

- Multiple Requests

- WebSocket

- File Download, Upload

- HTTP/2 Server Push

HTTP 2 is still like HTTP 1.1 a. request, response based, b. GET/POS/DELETE, … methods

Difference: HTTP 2.0 – a. Binary protocol, Mandatory TLS, Multiplexing over single TCP conn. (streams), Server push capability

Java Http Client Libraries

- Java 1.1 HttpURLConnection
- Apache HttpComponents
- Square's OkHttp
- Retrofit built up-on OkHttp
- JAX-RS REST Client
- Jetty Http Client
- AHC - Asyn Http Client
- Google HTTP Java Client
- . . .
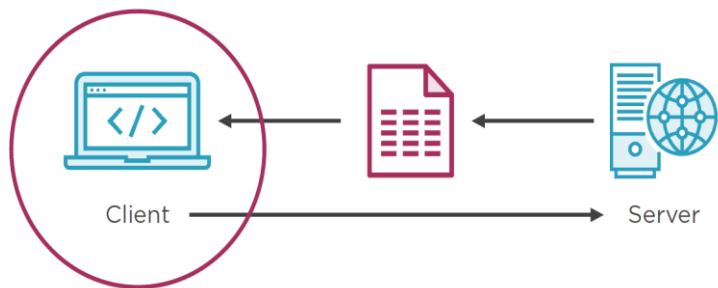
	Java version compatibility (current version of client)	Original release date	Latest release (as of September 2020)	Sync/Async	Async API style(s)	HTTP/2	Forms	Multipart / file upload	Cookies	Authentication	Transparent content compression	Caching	Websockets
HttpURLConnection	1.1+ (HTTP) 1.4+ (HTTPS)	1997 (with JDK 1.1) 2002 (with JDK 1.4)	September 2020 (with Java SE 15)	Sync only	N/A	No	No	No	No	None	No	No	No
Java HttpClient	9+ (as incubator module) 11+ (GA)	September 2018 (with Java SE 11)	September 2020 (with Java SE 15)	Both	Futures	Yes	No	No	Yes	Basic (not pre-emptive) Pluggable	No	No	Yes
Apache HTTPClient	7+	2001	September 2020	Both	Futures	Currently only in the 5.1 beta	Yes	Yes	Yes	Basic Digest NTLM SPNEGO Kerberos Pluggable	GZip Deflate	Yes	No
OkHttp	8+	2013	September 2020	Both	Callbacks	Yes	Yes	Yes	Yes	Pluggable	GZip Deflate Brotli	Yes	Yes
AsyncHttpClient	8+	2010	April 2020	Async only	Futures Callbacks Reactive streams	No	Yes	Yes	Yes	None	No	Yes	Yes
Jetty HttpClient	8+	2009	July 2020	Both	Callbacks	Yes	Yes	Yes	Yes	Basic Digest SPNEGO Pluggable	GZip	No	Yes

Java HttpURLConnection

What's Wrong with HttpURLConnection?

➢ Designed for HTTP/1.1
➢ Since Java 1.1 so you can't use those benefits from Generics, Enums, Lambdas
➢ You need to manage the edge cases, close connection etc
➢ Casting URLConnection
➢ Method names are string, e.g. GET, no ENUMs in Java 1.1.
➢ Returns raw input stream, need to decorate it for readability
➢ http client HttpUrlConnection doesn't support HTTP/2. Use OkHttp or Java11-http client
➢ Before Java 11: Apache HttpComponents, Square's OkHttp, JAX-RS REST Client

```java
public static void main(String[] args) throws Exception {
URL oracle = new URL("https://docs.oracle.com/javase/tutorial/networking/urls/readingWriting.html");
HttpURLConnection conn = (HttpURLConnection) oracle.openConnection(); //Casting
conn.setRequestMethod("GET"); //No ENUM
conn.setRequestProperty("User-Agent", "Java 1.1");

BufferedReader in = new BufferedReader(new InputStreamReader(conn.getInputStream())); //low-level
String inputLine;
. . .
```
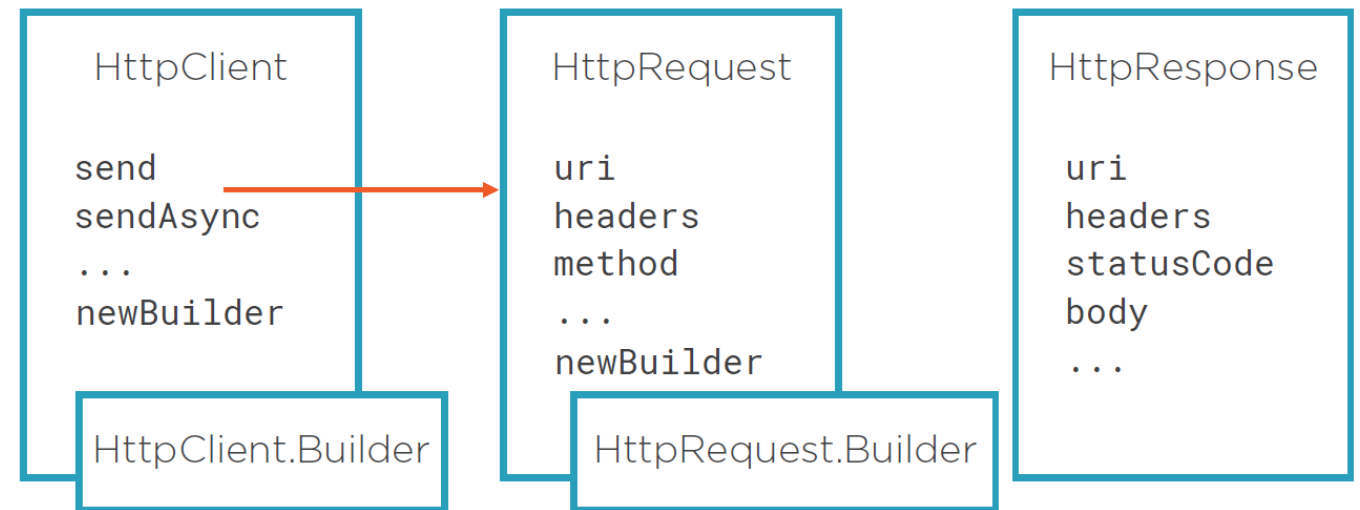
An_Old_HttpUrlConnection.java

Introducing HttpClient

✓ Introduced in Java 11 (easy and powerful)
✓ Replaces HttpURLConnection API
✓ Supports HTTP/2, WebSocket, Server Push, etc.
✓ Sync & async methods
✓ HttpClient Module – Java 9 modularity
✓ ???

module-info.java

```
module myapplication {
    requires java.net.http;
}
```

HttpClient: Important Types

HttpClient	HttpRequest	HttpResponse
send	uri	uri
sendAsync	headers	headers
...	method	statusCode
newBuilder	...	body
	newBuilder	...
HttpClient.Builder	HttpRequest.Builder	

java.net.http

BodyHandler

- ❖ ofString() String
- ❖ ofByteArray() byte[]
- ❖ ofFile(Path) Path
- ❖ ofLines() Stream<String>
- ❖ discarding() Void

Synch-Async Requests

```
public abstract <T> HttpResponse<T>
    send(HttpRequest request, HttpResponse.BodyHandler<T> responseBodyHandler)
        throws IOException, InterruptedException;


public abstract <T> CompletableFuture<HttpResponse<T>>
    sendAsync(HttpRequest request, BodyHandler<T> responseBodyHandler);
```

CompletableFuture<HttpResponse>   CompletableFuture<String>   CompletableFuture<Integer>

.thenApply(HttpResponse::body)   .thenApply(String::length)

completableFutureIntegerResponse.thenAccept(System.*out::println*).join(); *//or get()*

HttpClient Configuration

➢ Configuration affects all requests

➢ Create multiple instances when necessary

➢ Default Settings:

The default settings include: <code> prefer HTTP/2,
no connection timeout, redirection policy of NEVER, no cookie handler,
no authenticator, default thread pool executor, default proxy selector,
default SSL context

➢ HTTP Version (HTTP_1_1, HTTP_2)

➢ Priority

(Only affects HTTP/2 requests, Range 1-256 (inclusive))

➢ Redirection –

e.g .followRedirects(Redirect.NORMAL, NEVER, ALWAYS)

➢ Default Connection Timeout - .connectTimeout(Duration.ofSeconds(3))

[Not confuse with request timeout!]

➢ Custom Executor - **.executor(exec)**

Custom settings e.g.

```
var client = HttpClient.newBuilder()
.authenticator(Authenticator.getDefault())
.connectTimeout(Duration.ofSeconds(30))
.cookieHandler(CookieHandler.getDefault())
.executor(Executors.newFixedThreadPool(2))
.followRedirects(Redirect.NEVER)
.connectTimeout(Duration.ofSeconds(5))
.priority(1)
.proxy(ProxySelector.getDefault())
.sslContext(SSLContext.getDefault())
.version(Version.HTTP_2)
.sslParameters(new SSLParameters())
.build();
```

BodyPublisher

POST, PUT, PATCH, method - used as a PAYLOAD as being BODY PUBLISHER

I HttpClientBodyPublishers

POST(BodyPublisher publisher)
PUT(BodyPublisher publisher)
method(String method, BodyPublisher publisher)

Pre-defined BodyPublishers

BodyPublisher is a sub-interface of **Flow.Publisher**, introduced in Java 9.
Similarly, **BodySubscriber** is a sub-interface of **Flow.Subscriber**.
This means that these interfaces are aligned with the **reactive streams approach**,
which is suitable for asynchronously sending requests using HTTP/2.

➤ ofString(String body)
➤ ofByteArray(byte[] buf)
➤ ofFile(Path p)
➤ ofInputStream(Supplier<? extends InputStream> s)
➤ noBody()

Request with Body

```
HttpRequest.newBuilder(URI.create("..."))
           .POST(BodyPublishers.ofString("payload"))
           .build()
```

Headers & Cookies

Cookies – are persistent, used to manage state, token, etc …

HttpClient.Builder::setCookieHandler(CookieHandler handler)

```
CookieManager cookieManager = new CookieManager(null,
CookiePolicy.ACCEPT_ALL);
httpClient = HttpClient.newBuilder().cookieHandler(cookieManager).build();
```

```
//CookieStore: Default in-memory implementation
System.out.println(cookieManager.getCookieStore().getCookies());
```

```
GET /index.html

Set-Cookie: <name>=<value>

Cookie: <name>=<value>
```
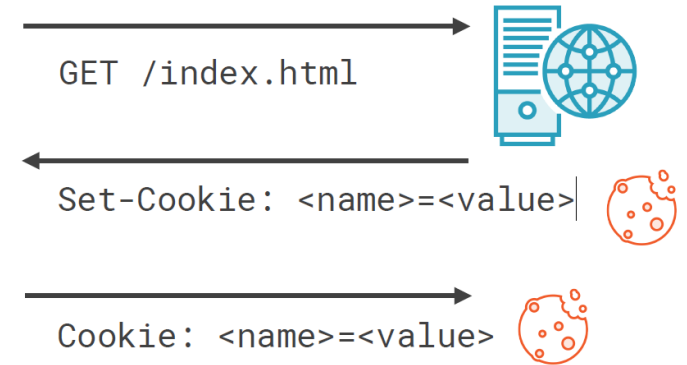
Headers - e.g. host-name is mandatory, we pass it automatically, …

```
HttpRequest httpRequest = HttpRequest.newBuilder(URI.create("http://jsonvat.com/"))
.header("Accept", "text/html").header("content-language", "en-us")
.headers("User-Agent", "Java", "Cache-Control", "no-transform", "Cache-Control", "no-store")
// to replace header
.setHeader("Accept", "application/json")
.POST(BodyPublishers.ofString("some body infor form"))
.build();
```

J_HttpClientCookiesAndHeadersDemo

Security
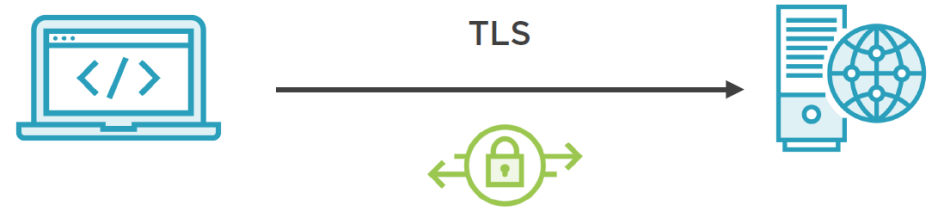
Secure Connections

- Default set of root Certification Authority (CA) certificates[public keys] since Java 10.

  >C:\apps\Java\jdk-11\lib\security > keytool -list -keystore cacerts >changeit

- **Self-signed certificates** (not in above list) add your CA to **trust store (1)**

  >$JAVA_HOME/bin/keytool -Djavax.net.ssl.trustStore /path/to/truststore

- **Mutual authentication** (Add client certs[private keys] to a **key store (2)**)

  >$JAVA_HOME/bin/keytool -Djavax.net.ssl.keystore /path/to/keystore

SSLContext - Create custom SSLContext with trust store (1) / keystore (2)

SSLParameters - TLS version, algorithms, etc.

```
//Create custom SSLContext
SSLContext sslCtx = SSLContext.getDefault();
/* TLSv1.2 or TLSv1.3 */
SSLParameters sslParameters =
    new SSLParameters(new String[]
{ "TLSv1.3" },
new String[] {"TLS_AES_128_GCM_SHA256" });
HttpClient httpClient = HttpClient.newBuilder()
.sslContext(sslCtx).sslParameters(sslParameters)
.build();
```

HTTP Basic Authentication   (e.g. Usages in Servlets, Spring) credentials == base64(user + ":" + password)

1. Use Authenticator 2. Proxy - also used for security purposes as well

```
Authenticator authenticator = new Authenticator() {
        protected PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication("admin", "secret".toCharArray()); }
};
// ProxySelector proxySelector = ProxySelector.getDefault();
ProxySelector proxySelector = ProxySelector.of(new InetSocketAddress("company.proxyserver.com", 8080));
HttpClient httpClient = HttpClient.newBuilder().authenticator(authenticator).proxy(proxySelector).build();
```

Advanced HttpClient Features - WebSocket

The `java.net.http` module also contains a client for WebSocket communication

- Real time, Message-based protocol
- Full-duplex, bi-directional communication
- Text and binary
- URI schemes - ws, wss

WebSocket.Listener – has default methods, just implement one you need

onOpen
onClose
onError
onText
onBinary
onPing
onPong

```java
CompletableFuture<WebSocket> wsFuture =
    HttpClient.newHttpClient()
            .newWebSocketBuilder()
            .buildAsync(
                URI.create("ws://server-url"),
                webSocketListener
            );
```

Application Layer
[HTTP, WebSocket]

Presentation Layer
[SSL]

Session Layer
[Sockets]

Transport Layer
[TCP, UDP]]

Network Layer

Data Link Layer

Physical Layer

Java 9 brings support for reactive programming or distributed asynchronous programming via the **publish/subscribe protocol** that forms the basis of **Flow API —** SubmissionPublisher, Flow [Publisher, Subscriber, Subscription, and Processor]. Also Java 11 http2-client embraces the concurrency and reactive programming ideas.

Advanced HttpClient Features - HTTP/2 Server Push

Push Promise

PushPromiseHandler

HttpClient::sendAsync(HttpRequest request,
BodyHandler bodyHandler,
PushPromiseHandler pushHandler)

HTTP/2 Server Push allows an HTTP/2-compliant servers (Tomcat, Jetty, Node.js, Nginx, Wildfly, ..) to send resources to a HTTP/2-compliant client before the client requests them. Old way was: preloading of resources

Config in Spring Boot App: server.http2.enabled=true

//endpoint, which would be powered by HTTP/2 Push Technology

```
@GetMapping(path = "/serviceWithPush")
public String serviceWithPush(HttpServletRequest request,PushBuilder pushBuilder) {
if (null != pushBuilder) {
    pushBuilder.path("resources/OnlineJavaPapers.png")
        .push();
}
return "index";

}
```

//or via Servlet API

```
public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    final String titleText;
    var builder = req.newPushBuilder();
```
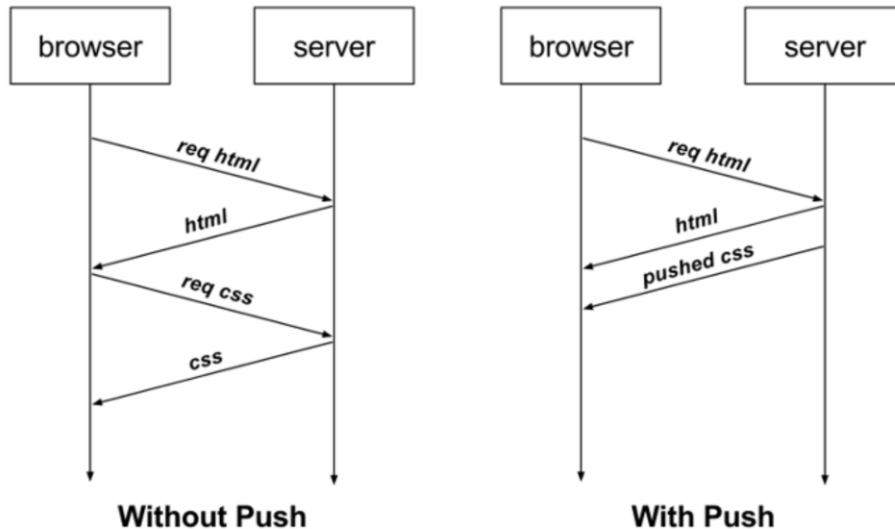
var resultMap = new ConcurrentHashMap<HttpRequest, CompletableFuture<HttpResponse<String>>>();
PushPromiseHandler<String> pph = PushPromiseHandler.of(pushPromise -> BodyHandlers.ofString(), resultMap);
client.sendAsync(request, BodyHandlers.*ofByteArray(), pph)*

Advanced HttpClient Features - Multiple Requests

L_MultipleRequests

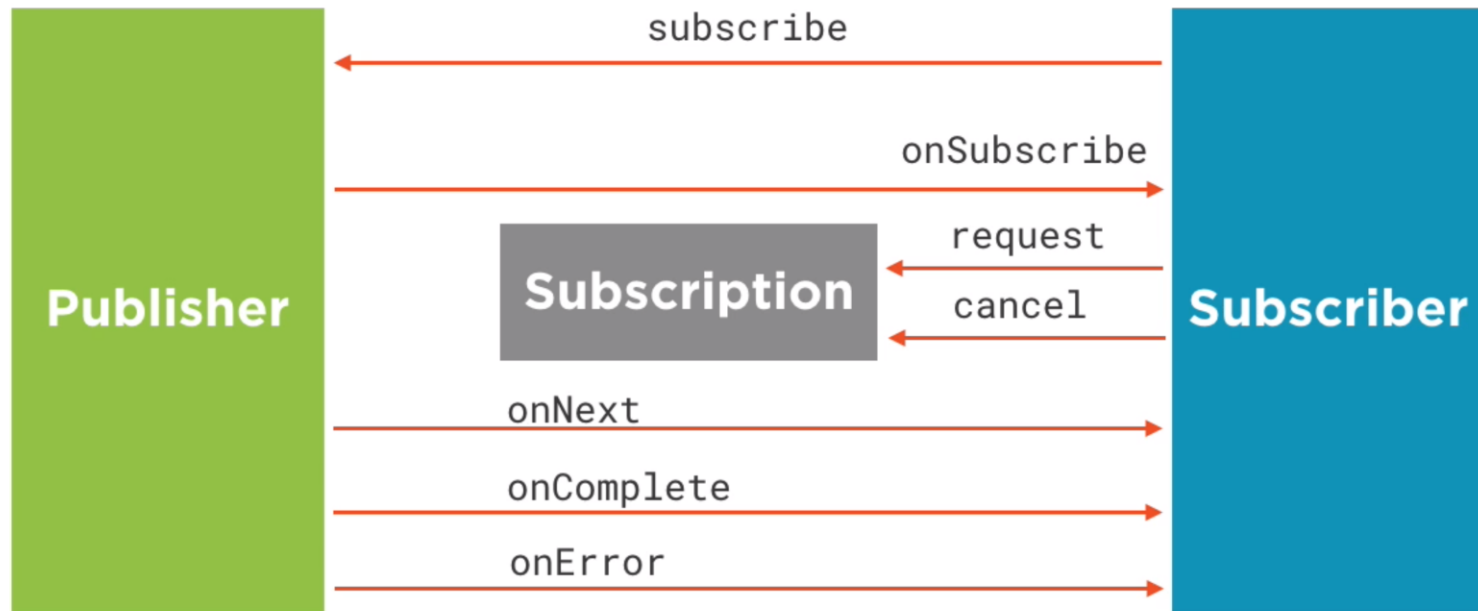# Advanced HttpClient Features – File Download/Upload

N_FileDownloadUpload

Flow API

Reactive Streams

Stream data with support for backpressure

Vendor-neutral specification (http://www.reactive-streams.org)

Flow API: interfaces added to JDK

Interoperability for reactive projects like RxJava, Akka Streams

Not meant as an end-user API

Adoption of java.util.concurrent.Flow

HttpClient implements Publisher/Subscriber interfaces

Following projects announced j.u.c.Flow support:

RxJava 2

Spring 5

Akka Streams

THANK YOU

References
https://openjdk.java.net/groups/net/httpclient/intro.html
https://factoryhr.medium.com/http-2-the-difference-between-http-1-1-benefits-and-how-to-use-it-38094fa0e95b
https://developer.ibm.com/languages/java/tutorials/java-theory-and-practice-3/