HttpCore Tutorial

Oleg Kalnichevski

Preface iv
1. HttpCore Scope iv
2. HttpCore Goals iv
3. What HttpCore is NOT iv
1. Fundamentals 1
1.1. HTTP messages 1
1.1.1. Structure 1
1.1.2. Basic operations 1
1.1.3. HTTP entity 3
1.1.4. Creating entities 5
1.2. HTTP protocol processors 7
1.2.1. Standard protocol interceptors 7
1.2.2. Working with protocol processors 8
1.3. HTTP execution context 9
1.3.1. Context sharing 9
2. Blocking I/O model 10
2.1. Blocking HTTP connections 10
2.1.1. Working with blocking HTTP connections 10
2.1.2. Content transfer with blocking I/O 11
2.1.3. Supported content transfer mechanisms 11
2.1.4. Terminating HTTP connections 12
2.2. HTTP exception handling 12
2.2.1. Protocol exception 12
2.3. Blocking HTTP protocol handlers 12
2.3.1. HTTP service 12
2.3.2. HTTP request executor 14
2.3.3. Connection persistence / re-use 14
2.4. Connection pools 15
2.5. TLS/SSL support 16
2.6. Embedded HTTP server 17
3. Asynchronous I/O based on NIO 18
3.1. Differences from other I/O frameworks 18
3.2. I/O reactor 18
3.2.1. I/O dispatchers 18
3.2.2. I/O reactor shutdown 19
3.2.3. I/O sessions 19
3.2.4. I/O session state management 19
3.2.5. I/O session event mask 19
3.2.6. I/O session buffers 20
3.2.7. I/O session shutdown 20
3.2.8. Listening I/O reactors 20
3.2.9. Connecting I/O reactors 21
3.3. I/O reactor configuration 22
3.3.1. Queuing of I/O interest set operations 23
3.4. I/O reactor exception handling 23
3.4.1. I/O reactor audit log 24
3.5. Non-blocking HTTP connections 24
3.5.1. Execution context of non-blocking HTTP connections 24
3.5.2. Working with non-blocking HTTP connections 24

3.5.3. HTTP I/O control 25
3.5.4. Non-blocking content transfer 26
3.5.5. Supported non-blocking content transfer mechanisms 27
3.5.6. Direct channel I/O 27
3.6. HTTP I/O event dispatchers 28
3.7. Non-blocking HTTP content producers 30
3.7.1. Creating non-blocking entities 30
3.8. Non-blocking HTTP protocol handlers 31
3.8.1. Asynchronous HTTP service 31
3.8.2. Asynchronous HTTP request executor 35
3.9. Non-blocking connection pools 37
3.10. Pipelined request execution 38
3.11. Non-blocking TLS/SSL 39
3.11.1. SSL I/O session 39
3.11.2. TLS/SSL aware I/O event dispatches 40
3.12. Embedded non-blocking HTTP server 41
4. Advanced topics 42
4.1. HTTP message parsing and formatting framework 42
4.1.1. HTTP line parsing and formatting 42
4.1.2. HTTP message streams and session I/O buffers 44
4.1.3. HTTP message parsers and formatters 45
4.1.4. HTTP header parsing on demand 47

Preface

HttpCore is a set of components implementing the most fundamental aspects of the HTTP protocol that are nonetheless sufficient to develop full-featured client-side and server-side HTTP services with a minimal footprint.

HttpCore has the following scope and goals:

1. HttpCore Scope

- A consistent API for building client / proxy / server side HTTP services
- A consistent API for building both synchronous and asynchronous HTTP services
- A set of low level components based on blocking (classic) and non-blocking (NIO) I/O models

2. HttpCore Goals

- Implementation of the most fundamental HTTP transport aspects
- Balance between good performance and the clarity & expressiveness of API
- Small (predictable) memory footprint
- Self-contained library (no external dependencies beyond JRE)

3. What HttpCore is NOT

- A replacement for HttpClient
- A replacement for Servlet APIs

Chapter 1. Fundamentals

1.1. HTTP messages

1.1.1. Structure

A HTTP message consists of a header and an optional body. The message header of an HTTP request consists of a request line and a collection of header fields. The message header of an HTTP response consists of a status line and a collection of header fields. All HTTP messages must include the protocol version. Some HTTP messages can optionally enclose a content body.

HttpCore defines the HTTP message object model to follow this definition closely, and provides extensive support for serialization (formatting) and deserialization (parsing) of HTTP message elements.

1.1.2. Basic operations

1.1.2.1. HTTP request message

HTTP request is a message sent from the client to the server. The first line of that message includes the method to apply to the resource, the identifier of the resource, and the protocol version in use.

```
HttpRequest request = new BasicHttpRequest("GET", "/",
    HttpVersion.HTTP_1_1);

System.out.println(request.getRequestLine().getMethod());
System.out.println(request.getRequestLine().getUri());
System.out.println(request.getProtocolVersion());
System.out.println(request.getRequestLine().toString());
```

stdout >

```
GET
/
HTTP/1.1
GET / HTTP/1.1
```

1.1.2.2. HTTP response message

HTTP response is a message sent by the server back to the client after having received and interpreted a request message. The first line of that message consists of the protocol version followed by a numeric status code and its associated textual phrase.

```
HttpResponse response = new BasicHttpResponse(HttpVersion.HTTP_1_1,
    HttpStatus.SC_OK, "OK");

System.out.println(response.getProtocolVersion());
System.out.println(response.getStatusLine().getStatusCode());
System.out.println(response.getStatusLine().getReasonPhrase());
System.out.println(response.getStatusLine().toString());
```

stdout >

```
HTTP/1.1
200
OK
HTTP/1.1 200 OK
```

1.1.2.3. HTTP message common properties and methods

An HTTP message can contain a number of headers describing properties of the message such as the content length, content type, and so on. HttpCore provides methods to retrieve, add, remove, and enumerate such headers.

```
HttpResponse response = new BasicHttpResponse(HttpVersion.HTTP_1_1,
    HttpStatus.SC_OK, "OK");
response.addHeader("Set-Cookie",
    "c1=a; path=/; domain=localhost");
response.addHeader("Set-Cookie",
    "c2=b; path=\"/\", c3=c; domain=\"localhost\"");
Header h1 = response.getFirstHeader("Set-Cookie");
System.out.println(h1);
Header h2 = response.getLastHeader("Set-Cookie");
System.out.println(h2);
Header[] hs = response.getHeaders("Set-Cookie");
System.out.println(hs.length);
```

stdout >

```
Set-Cookie: c1=a; path=/; domain=localhost
Set-Cookie: c2=b; path="/", c3=c; domain="localhost"
2
```

There is an efficient way to obtain all headers of a given type using the `HeaderIterator` interface.

```
HttpResponse response = new BasicHttpResponse(HttpVersion.HTTP_1_1,
    HttpStatus.SC_OK, "OK");
response.addHeader("Set-Cookie",
    "c1=a; path=/; domain=localhost");
response.addHeader("Set-Cookie",
    "c2=b; path=\"/\", c3=c; domain=\"localhost\"");

HeaderIterator it = response.headerIterator("Set-Cookie");

while (it.hasNext()) {
    System.out.println(it.next());
}
```

stdout >

```
Set-Cookie: c1=a; path=/; domain=localhost
Set-Cookie: c2=b; path="/", c3=c; domain="localhost"
```

It also provides convenience methods to parse HTTP messages into individual header elements.

```
HttpResponse response = new BasicHttpResponse(HttpVersion.HTTP_1_1,
    HttpStatus.SC_OK, "OK");
response.addHeader("Set-Cookie",
    "c1=a; path=/; domain=localhost");
```

```
response.addHeader("Set-Cookie",
    "c2=b; path=\"/\", c3=c; domain=\"localhost\"");

HeaderElementIterator it = new BasicHeaderElementIterator(
        response.headerIterator("Set-Cookie"));

while (it.hasNext()) {
    HeaderElement elem = it.nextElement();
    System.out.println(elem.getName() + " = " + elem.getValue());
    NameValuePair[] params = elem.getParameters();
    for (int i = 0; i < params.length; i++) {
        System.out.println(" " + params[i]);
    }
}
```

stdout >

```
c1 = a
 path=/
 domain=localhost
c2 = b
 path=/
c3 = c
 domain=localhost
```

HTTP headers are tokenized into individual header elements only on demand. HTTP headers received over an HTTP connection are stored internally as an array of characters and parsed lazily only when you access their properties.

1.1.3. HTTP entity

HTTP messages can carry a content entity associated with the request or response. Entities can be found in some requests and in some responses, as they are optional. Requests that use entities are referred to as entity-enclosing requests. The HTTP specification defines two entity-enclosing methods: POST and PUT. Responses are usually expected to enclose a content entity. There are exceptions to this rule such as responses to HEAD method and 204 No Content, 304 Not Modified, 205 Reset Content responses.

HttpCore distinguishes three kinds of entities, depending on where their content originates:

- **streamed:** The content is received from a stream, or generated on the fly. In particular, this category includes entities being received from a connection. Streamed entities are generally not repeatable.
- **self-contained:** The content is in memory or obtained by means that are independent from a connection or other entity. Self-contained entities are generally repeatable.
- **wrapping:** The content is obtained from another entity.

1.1.3.1. Repeatable entities

An entity can be repeatable, meaning its content can be read more than once. This is only possible with self-contained entities (like `ByteArrayEntity` or `StringEntity`).

1.1.3.2. Using HTTP entities

Since an entity can represent both binary and character content, it has support for character encodings (to support the latter, i.e. character content).

The entity is created when executing a request with enclosed content or when the request was successful and the response body is used to send the result back to the client.

To read the content from the entity, one can either retrieve the input stream via the `HttpEntity#getContent()` method, which returns an `java.io.InputStream`, or one can supply an output stream to the `HttpEntity#writeTo(OutputStream)` method, which will return once all content has been written to the given stream. Please note that some non-streaming (self-contained) entities may be unable to represent their content as a `java.io.InputStream` efficiently. It is legal for such entities to implement `HttpEntity#writeTo(OutputStream)` method only and to throw `UnsupportedOperationException` from `HttpEntity#getContent()` method.

The `EntityUtils` class exposes several static methods to simplify extracting the content or information from an entity. Instead of reading the `java.io.InputStream` directly, one can retrieve the complete content body in a string or byte array by using the methods from this class.

When the entity has been received with an incoming message, the methods `HttpEntity#getContentType()` and `HttpEntity#getContentLength()` methods can be used for reading the common metadata such as `Content-Type` and `Content-Length` headers (if they are available). Since the `Content-Type` header can contain a character encoding for text mime-types like `text/plain` or `text/html`, the `HttpEntity#getContentEncoding()` method is used to read this information. If the headers aren't available, a length of -1 will be returned, and `NULL` for the content type. If the `Content-Type` header is available, a Header object will be returned.

When creating an entity for a outgoing message, this meta data has to be supplied by the creator of the entity.

```
StringEntity myEntity = new StringEntity("important message",
    Consts.UTF_8);

System.out.println(myEntity.getContentType());
System.out.println(myEntity.getContentLength());
System.out.println(EntityUtils.toString(myEntity));
System.out.println(EntityUtils.toByteArray(myEntity).length);
```

stdout >

```
Content-Type: text/plain; charset=UTF-8
17
important message
17
```

1.1.3.3. Ensuring release of system resources

In order to ensure proper release of system resources one must close the content stream associated with the entity.

```
HttpResponse response;
HttpEntity entity = response.getEntity();
if (entity != null) {
    InputStream instream = entity.getContent();
    try {
        // do something useful
    } finally {
        instream.close();
    }
```

```
}
```

When working with streaming entities, one can use the `EntityUtils#consume(HttpEntity)` method to ensure that the entity content has been fully consumed and the underlying stream has been closed.

1.1.4. Creating entities

There are a few ways to create entities. HttpCore provides the following implementations:

- `BasicHttpEntity`
- `ByteArrayEntity`
- `StringEntity`
- `InputStreamEntity`
- `FileEntity`
- `EntityTemplate`
- `HttpEntityWrapper`
- `BufferedHttpEntity`

1.1.4.1. `BasicHttpEntity`

Exactly as the name implies, this basic entity represents an underlying stream. In general, use this class for entities received from HTTP messages.

This entity has an empty constructor. After construction, it represents no content, and has a negative content length.

One needs to set the content stream, and optionally the length. This can be done with the `BasicHttpEntity#setContent(InputStream)` and `BasicHttpEntity#setContentLength(long)` methods respectively.

```
BasicHttpEntity myEntity = new BasicHttpEntity();
myEntity.setContent(someInputStream);
myEntity.setContentLength(340); // sets the length to 340
```

1.1.4.2. `ByteArrayEntity`

`ByteArrayEntity` is a self-contained, repeatable entity that obtains its content from a given byte array. Supply the byte array to the constructor.

```
ByteArrayEntity myEntity = new ByteArrayEntity(new byte[] {1,2,3},
        ContentType.APPLICATION_OCTET_STREAM);
```

1.1.4.3. `StringEntity`

`StringEntity` is a self-contained, repeatable entity that obtains its content from a `java.lang.String` object. It has three constructors, one simply constructs with a given `java.lang.String` object; the second also takes a character encoding for the data in the string; the third allows the mime type to be specified.

```
StringBuilder sb = new StringBuilder();
Map<String, String> env = System.getenv();
for (Map.Entry<String, String> envEntry : env.entrySet()) {
    sb.append(envEntry.getKey())
            .append(": ").append(envEntry.getValue())
            .append("\r\n");
}

// construct without a character encoding (defaults to ISO-8859-1)
HttpEntity myEntity1 = new StringEntity(sb.toString());

// alternatively construct with an encoding (mime type defaults to "text/plain")
HttpEntity myEntity2 = new StringEntity(sb.toString(), Consts.UTF_8);

// alternatively construct with an encoding and a mime type
HttpEntity myEntity3 = new StringEntity(sb.toString(),
        ContentType.create("text/plain", Consts.UTF_8));
```

1.1.4.4. `InputStreamEntity`

`InputStreamEntity` is a streamed, non-repeatable entity that obtains its content from an input stream. Construct it by supplying the input stream and the content length. Use the content length to limit the amount of data read from the `java.io.InputStream`. If the length matches the content length available on the input stream, then all data will be sent. Alternatively, a negative content length will read all data from the input stream, which is the same as supplying the exact content length, so use the length to limit the amount of data to read.

```
InputStream instream = getSomeInputStream();
InputStreamEntity myEntity = new InputStreamEntity(instream, 16);
```

1.1.4.5. `FileEntity`

`FileEntity` is a self-contained, repeatable entity that obtains its content from a file. Use this mostly to stream large files of different types, where you need to supply the content type of the file, for instance, sending a zip file would require the content type `application/zip`, for XML `application/xml`.

```
HttpEntity entity = new FileEntity(staticFile,
        ContentType.create("application/java-archive"));
```

1.1.4.6. `HttpEntityWrapper`

This is the base class for creating wrapped entities. The wrapping entity holds a reference to a wrapped entity and delegates all calls to it. Implementations of wrapping entities can derive from this class and need to override only those methods that should not be delegated to the wrapped entity.

1.1.4.7. `BufferedHttpEntity`

`BufferedHttpEntity` is a subclass of `HttpEntityWrapper`. Construct it by supplying another entity. It reads the content from the supplied entity, and buffers it in memory.

This makes it possible to make a repeatable entity, from a non-repeatable entity. If the supplied entity is already repeatable, it simply passes calls through to the underlying entity.

```
myNonRepeatableEntity.setContent(someInputStream);
BufferedHttpEntity myBufferedEntity = new BufferedHttpEntity(
```

```
  myNonRepeatableEntity);
```

1.2. HTTP protocol processors

HTTP protocol interceptor is a routine that implements a specific aspect of the HTTP protocol. Usually protocol interceptors are expected to act upon one specific header or a group of related headers of the incoming message or populate the outgoing message with one specific header or a group of related headers. Protocol interceptors can also manipulate content entities enclosed with messages; transparent content compression / decompression being a good example. Usually this is accomplished by using the 'Decorator' pattern where a wrapper entity class is used to decorate the original entity. Several protocol interceptors can be combined to form one logical unit.

HTTP protocol processor is a collection of protocol interceptors that implements the 'Chain of Responsibility' pattern, where each individual protocol interceptor is expected to work on the particular aspect of the HTTP protocol it is responsible for.

Usually the order in which interceptors are executed should not matter as long as they do not depend on a particular state of the execution context. If protocol interceptors have interdependencies and therefore must be executed in a particular order, they should be added to the protocol processor in the same sequence as their expected execution order.

Protocol interceptors must be implemented as thread-safe. Similarly to servlets, protocol interceptors should not use instance variables unless access to those variables is synchronized.

1.2.1. Standard protocol interceptors

HttpCore comes with a number of most essential protocol interceptors for client and server HTTP processing.

1.2.1.1. `RequestContent`

`RequestContent` is the most important interceptor for outgoing requests. It is responsible for delimiting content length by adding the `Content-Length` or `Transfer-Content` headers based on the properties of the enclosed entity and the protocol version. This interceptor is required for correct functioning of client side protocol processors.

1.2.1.2. `ResponseContent`

`ResponseContent` is the most important interceptor for outgoing responses. It is responsible for delimiting content length by adding `Content-Length` or `Transfer-Content` headers based on the properties of the enclosed entity and the protocol version. This interceptor is required for correct functioning of server side protocol processors.

1.2.1.3. `RequestConnControl`

`RequestConnControl` is responsible for adding the `Connection` header to the outgoing requests, which is essential for managing persistence of `HTTP/1.0` connections. This interceptor is recommended for client side protocol processors.

1.2.1.4. `ResponseConnControl`

`ResponseConnControl` is responsible for adding the `Connection` header to the outgoing responses, which is essential for managing persistence of `HTTP/1.0` connections. This interceptor is recommended for server side protocol processors.

1.2.1.5. `RequestDate`

`RequestDate` is responsible for adding the `Date` header to the outgoing requests. This interceptor is optional for client side protocol processors.

1.2.1.6. `ResponseDate`

`ResponseDate` is responsible for adding the `Date` header to the outgoing responses. This interceptor is recommended for server side protocol processors.

1.2.1.7. `RequestExpectContinue`

`RequestExpectContinue` is responsible for enabling the 'expect-continue' handshake by adding the `Expect` header. This interceptor is recommended for client side protocol processors.

1.2.1.8. `RequestTargetHost`

`RequestTargetHost` is responsible for adding the `Host` header. This interceptor is required for client side protocol processors.

1.2.1.9. `RequestUserAgent`

`RequestUserAgent` is responsible for adding the `User-Agent` header. This interceptor is recommended for client side protocol processors.

1.2.1.10. `ResponseServer`

`ResponseServer` is responsible for adding the `Server` header. This interceptor is recommended for server side protocol processors.

1.2.2. Working with protocol processors

Usually HTTP protocol processors are used to pre-process incoming messages prior to executing application specific processing logic and to post-process outgoing messages.

```
HttpProcessor httpproc = HttpProcessorBuilder.create()
        // Required protocol interceptors
        .add(new RequestContent())
        .add(new RequestTargetHost())
        // Recommended protocol interceptors
        .add(new RequestConnControl())
        .add(new RequestUserAgent("MyAgent-HTTP/1.1"))
        // Optional protocol interceptors
        .add(new RequestExpectContinue(true))
        .build();

HttpCoreContext context = HttpCoreContext.create();
HttpRequest request = new BasicHttpRequest("GET", "/");
httpproc.process(request, context);
```

Send the request to the target host and get a response.

```
HttpResponse = <...>
httpproc.process(response, context);
```

Please note the `BasicHttpProcessor` class does not synchronize access to its internal structures and therefore may not be thread-safe.

1.3. HTTP execution context

Originally HTTP has been designed as a stateless, response-request oriented protocol. However, real world applications often need to be able to persist state information through several logically related request-response exchanges. In order to enable applications to maintain a processing state HttpCpre allows HTTP messages to be executed within a particular execution context, referred to as HTTP context. Multiple logically related messages can participate in a logical session if the same context is reused between consecutive requests. HTTP context functions similarly to a `java.util.Map<String, Object>`. It is simply a collection of logically related named values.

Please nore `HttpContext` can contain arbitrary objects and therefore may be unsafe to share between multiple threads. Care must be taken to ensure that `HttpContext` instances can be accessed by one thread at a time.

1.3.1. Context sharing

Protocol interceptors can collaborate by sharing information - such as a processing state - through an HTTP execution context. HTTP context is a structure that can be used to map an attribute name to an attribute value. Internally HTTP context implementations are usually backed by a `HashMap`. The primary purpose of the HTTP context is to facilitate information sharing among various logically related components. HTTP context can be used to store a processing state for one message or several consecutive messages. Multiple logically related messages can participate in a logical session if the same context is reused between consecutive messages.

```
HttpProcessor httpproc = HttpProcessorBuilder.create()
        .add(new HttpRequestInterceptor() {
            public void process(
                    HttpRequest request,
                    HttpContext context) throws HttpException, IOException {
                String id = (String) context.getAttribute("session-id");
                if (id != null) {
                    request.addHeader("Session-ID", id);
                }
            }
        })
        .build();

HttpCoreContext context = HttpCoreContext.create();
HttpRequest request = new BasicHttpRequest("GET", "/");
httpproc.process(request, context);
```

Chapter 2. Blocking I/O model

Blocking (or classic) I/O in Java represents a highly efficient and convenient I/O model well suited for high performance applications where the number of concurrent connections is relatively moderate. Modern JVMs are capable of efficient context switching and the blocking I/O model should offer the best performance in terms of raw data throughput as long as the number of concurrent connections is below one thousand and connections are mostly busy transmitting data. However for applications where connections stay idle most of the time the overhead of context switching may become substantial and a non-blocking I/O model may present a better alternative.

2.1. Blocking HTTP connections

HTTP connections are responsible for HTTP message serialization and deserialization. One should rarely need to use HTTP connection objects directly. There are higher level protocol components intended for execution and processing of HTTP requests. However, in some cases direct interaction with HTTP connections may be necessary, for instance, to access properties such as the connection status, the socket timeout or the local and remote addresses.

It is important to bear in mind that HTTP connections are not thread-safe. We strongly recommend limiting all interactions with HTTP connection objects to one thread. The only method of `HttpConnection` interface and its sub-interfaces which is safe to invoke from another thread is `HttpConnection#shutdown()` .

2.1.1. Working with blocking HTTP connections

HttpCore does not provide full support for opening connections because the process of establishing a new connection - especially on the client side - can be very complex when it involves one or more authenticating or/and tunneling proxies. Instead, blocking HTTP connections can be bound to any arbitrary network socket.

```
Socket socket = <...>

DefaultBHttpClientConnection conn = new DefaultBHttpClientConnection(8 * 1024);
conn.bind(socket);
System.out.println(conn.isOpen());
HttpConnectionMetrics metrics = conn.getMetrics();
System.out.println(metrics.getRequestCount());
System.out.println(metrics.getResponseCount());
System.out.println(metrics.getReceivedBytesCount());
System.out.println(metrics.getSentBytesCount());
```

HTTP connection interfaces, both client and server, send and receive messages in two stages. The message head is transmitted first. Depending on properties of the message head, a message body may follow it. Please note it is very important to always close the underlying content stream in order to signal that the processing of the message is complete. HTTP entities that stream out their content directly from the input stream of the underlying connection must ensure they fully consume the content of the message body for that connection to be potentially re-usable.

Over-simplified process of request execution on the client side may look like this:

```
Socket socket = <...>
```

```
DefaultBHttpClientConnection conn = new DefaultBHttpClientConnection(8 * 1024);
conn.bind(socket);
HttpRequest request = new BasicHttpRequest("GET", "/");
conn.sendRequestHeader(request);
HttpResponse response = conn.receiveResponseHeader();
conn.receiveResponseEntity(response);
HttpEntity entity = response.getEntity();
if (entity != null) {
    // Do something useful with the entity and, when done, ensure all
    // content has been consumed, so that the underlying connection
    // can be re-used
    EntityUtils.consume(entity);
}
```

Over-simplified process of request handling on the server side may look like this:

```

Socket socket = <...>

DefaultBHttpServerConnection conn = new DefaultBHttpServerConnection(8 * 1024);
conn.bind(socket);
HttpRequest request = conn.receiveRequestHeader();
if (request instanceof HttpEntityEnclosingRequest) {
    conn.receiveRequestEntity((HttpEntityEnclosingRequest) request);
    HttpEntity entity = ((HttpEntityEnclosingRequest) request)
            .getEntity();
    if (entity != null) {
        // Do something useful with the entity and, when done, ensure all
        // content has been consumed, so that the underlying connection
        // could be re-used
        EntityUtils.consume(entity);
    }
}
HttpResponse response = new BasicHttpResponse(HttpVersion.HTTP_1_1,
        200, "OK") ;
response.setEntity(new StringEntity("Got it") );
conn.sendResponseHeader(response);
conn.sendResponseEntity(response);
```

Please note that one should rarely need to transmit messages using these low level methods and should normally use the appropriate higher level HTTP service implementations instead.

2.1.2. Content transfer with blocking I/O

HTTP connections manage the process of the content transfer using the `HttpEntity` interface. HTTP connections generate an entity object that encapsulates the content stream of the incoming message. Please note that `HttpServerConnection#receiveRequestEntity()` and `HttpClientConnection#receiveResponseEntity()` do not retrieve or buffer any incoming data. They merely inject an appropriate content codec based on the properties of the incoming message. The content can be retrieved by reading from the content input stream of the enclosed entity using `HttpEntity#getContent()`. The incoming data will be decoded automatically and completely transparently to the data consumer. Likewise, HTTP connections rely on `HttpEntity#writeTo(OutputStream)` method to generate the content of an outgoing message. If an outgoing message encloses an entity, the content will be encoded automatically based on the properties of the message.

2.1.3. Supported content transfer mechanisms

Default implementations of HTTP connections support three content transfer mechanisms defined by the HTTP/1.1 specification:

- **`Content-Length` delimited:** The end of the content entity is determined by the value of the `Content-Length` header. Maximum entity length: `Long#MAX_VALUE`.

- **Identity coding:** The end of the content entity is demarcated by closing the underlying connection (end of stream condition). For obvious reasons the identity encoding can only be used on the server side. Maximum entity length: unlimited.

- **Chunk coding:** The content is sent in small chunks. Maximum entity length: unlimited.

The appropriate content stream class will be created automatically depending on properties of the entity enclosed with the message.

2.1.4. Terminating HTTP connections

HTTP connections can be terminated either gracefully by calling `HttpConnection#close()` or forcibly by calling `HttpConnection#shutdown()`. The former tries to flush all buffered data prior to terminating the connection and may block indefinitely. The `HttpConnection#close()` method is not thread-safe. The latter terminates the connection without flushing internal buffers and returns control to the caller as soon as possible without blocking for long. The `HttpConnection#shutdown()` method is thread-safe.

2.2. HTTP exception handling

All HttpCore components potentially throw two types of exceptions: `IOException` in case of an I/O failure such as socket timeout or an socket reset and `HttpException` that signals an HTTP failure such as a violation of the HTTP protocol. Usually I/O errors are considered non-fatal and recoverable, whereas HTTP protocol errors are considered fatal and cannot be automatically recovered from.

2.2.1. Protocol exception

`ProtocolException` signals a fatal HTTP protocol violation that usually results in an immediate termination of the HTTP message processing.

2.3. Blocking HTTP protocol handlers

2.3.1. HTTP service

`HttpService` is a server side HTTP protocol handler based on the blocking I/O model that implements the essential requirements of the HTTP protocol for the server side message processing as described by RFC 2616.

`HttpService` relies on `HttpProcessor` instance to generate mandatory protocol headers for all outgoing messages and apply common, cross-cutting message transformations to all incoming and outgoing messages, whereas HTTP request handlers are expected to take care of application specific content generation and processing.

```
HttpProcessor httpproc = HttpProcessorBuilder.create()
        .add(new ResponseDate())
        .add(new ResponseServer("MyServer-HTTP/1.1"))
        .add(new ResponseContent())
        .add(new ResponseConnControl())
```

```
        .build();
HttpService httpService = new HttpService(httpproc, null);
```

2.3.1.1. HTTP request handlers

The `HttpRequestHandler` interface represents a routine for processing of a specific group of HTTP requests. `HttpService` is designed to take care of protocol specific aspects, whereas individual request handlers are expected to take care of application specific HTTP processing. The main purpose of a request handler is to generate a response object with a content entity to be sent back to the client in response to the given request.

```
HttpRequestHandler myRequestHandler = new HttpRequestHandler() {

    public void handle(
            HttpRequest request,
            HttpResponse response,
            HttpContext context) throws HttpException, IOException {
        response.setStatusCode(HttpStatus.SC_OK);
        response.setEntity(
                new StringEntity("some important message",
                        ContentType.TEXT_PLAIN));
    }

};
```

2.3.1.2. Request handler resolver

HTTP request handlers are usually managed by a `HttpRequestHandlerResolver` that matches a request URI to a request handler. HttpCore includes a very simple implementation of the request handler resolver based on a trivial pattern matching algorithm: `HttpRequestHandlerRegistry` supports only three formats: `*`, `<uri>*` and `*<uri>`.

```
HttpProcessor httpproc = <...>

HttpRequestHandler myRequestHandler1 = <...>
HttpRequestHandler myRequestHandler2 = <...>
HttpRequestHandler myRequestHandler3 = <...>

UriHttpRequestHandlerMapper handlerMapper = new UriHttpRequestHandlerMapper();
handlerMapper.register("/service/*", myRequestHandler1);
handlerMapper.register("*.do", myRequestHandler2);
handlerMapper.register("*", myRequestHandler3);
HttpService httpService = new HttpService(httpproc, handlerMapper);
```

Users are encouraged to provide more sophisticated implementations of `HttpRequestHandlerResolver` - for instance, based on regular expressions.

2.3.1.3. Using HTTP service to handle requests

When fully initialized and configured, the `HttpService` can be used to execute and handle requests for active HTTP connections. The `HttpService#handleRequest()` method reads an incoming request, generates a response and sends it back to the client. This method can be executed in a loop to handle multiple requests on a persistent connection. The `HttpService#handleRequest()` method is safe to execute from multiple threads. This allows processing of requests on several connections simultaneously, as long as all the protocol interceptors and requests handlers used by the `HttpService` are thread-safe.

```
HttpService httpService = <...>
HttpServerConnection conn = <...>
HttpContext context = <...>

boolean active = true;
try {
    while (active && conn.isOpen()) {
        httpService.handleRequest(conn, context);
    }
} finally {
    conn.shutdown();
}
```

2.3.2. HTTP request executor

`HttpRequestExecutor` is a client side HTTP protocol handler based on the blocking I/O model that implements the essential requirements of the HTTP protocol for the client side message processing, as described by RFC 2616. The `HttpRequestExecutor` relies on the `HttpProcessor` instance to generate mandatory protocol headers for all outgoing messages and apply common, cross-cutting message transformations to all incoming and outgoing messages. Application specific processing can be implemented outside `HttpRequestExecutor` once the request has been executed and a response has been received.

```
HttpClientConnection conn = <...>

HttpProcessor httpproc = HttpProcessorBuilder.create()
        .add(new RequestContent())
        .add(new RequestTargetHost())
        .add(new RequestConnControl())
        .add(new RequestUserAgent("MyClient/1.1"))
        .add(new RequestExpectContinue(true))
        .build();
HttpRequestExecutor httpexecutor = new HttpRequestExecutor();

HttpRequest request = new BasicHttpRequest("GET", "/");
HttpCoreContext context = HttpCoreContext.create();
httpexecutor.preProcess(request, httpproc, context);
HttpResponse response = httpexecutor.execute(request, conn, context);
httpexecutor.postProcess(response, httpproc, context);

HttpEntity entity = response.getEntity();
EntityUtils.consume(entity);
```

Methods of `HttpRequestExecutor` are safe to execute from multiple threads. This allows execution of requests on several connections simultaneously, as long as all the protocol interceptors used by the `HttpRequestExecutor` are thread-safe.

2.3.3. Connection persistence / re-use

The `ConnectionReuseStrategy` interface is intended to determine whether the underlying connection can be re-used for processing of further messages after the transmission of the current message has been completed. The default connection re-use strategy attempts to keep connections alive whenever possible. Firstly, it examines the version of the HTTP protocol used to transmit the message. `HTTP/1.1` connections are persistent by default, while `HTTP/1.0` connections are not. Secondly, it examines the value of the `Connection` header. The peer can indicate whether it intends to re-use the connection on the opposite side by sending `Keep-Alive` or `Close` values in the `Connection` header. Thirdly, the

strategy makes the decision whether the connection is safe to re-use based on the properties of the enclosed entity, if available.

2.4. Connection pools

Efficient client-side HTTP transports often requires effective re-use of persistent connections. HttpCore facilitates the process of connection re-use by providing support for managing pools of persistent HTTP connections. Connection pool implementations are thread-safe and can be used concurrently by multiple consumers.

By default the pool allows only 20 concurrent connections in total and two concurrent connections per a unique route. The two connection limit is due to the requirements of the HTTP specification. However, in practical terms this can often be too restrictive. One can change the pool configuration at runtime to allow for more concurrent connections depending on a particular application context.

```
HttpHost target = new HttpHost("localhost");
BasicConnPool connpool = new BasicConnPool();
connpool.setMaxTotal(200);
connpool.setDefaultMaxPerRoute(10);
connpool.setMaxPerRoute(target, 20);
Future<BasicPoolEntry> future = connpool.lease(target, null);
BasicPoolEntry poolEntry = future.get();
HttpClientConnection conn = poolEntry.getConnection();
```

Please note that the connection pool has no way of knowing whether or not a leased connection is still being used. It is the responsibility of the connection pool user to ensure that the connection is released back to the pool once it is not longer needed, even if the connection is not reusable.

```
BasicConnPool connpool = <...>
Future<BasicPoolEntry> future = connpool.lease(target, null);
BasicPoolEntry poolEntry = future.get();
try {
    HttpClientConnection conn = poolEntry.getConnection();
} finally {
    connpool.release(poolEntry, true);
}
```

The state of the connection pool can be interrogated at runtime.

```
HttpHost target = new HttpHost("localhost");
BasicConnPool connpool = <...>
PoolStats totalStats = connpool.getTotalStats();
System.out.println("total available: " + totalStats.getAvailable());
System.out.println("total leased: " + totalStats.getLeased());
System.out.println("total pending: " + totalStats.getPending());
PoolStats targetStats = connpool.getStats(target);
System.out.println("target available: " + targetStats.getAvailable());
System.out.println("target leased: " + targetStats.getLeased());
System.out.println("target pending: " + targetStats.getPending());
```

Please note that connection pools do not pro-actively evict expired connections. Even though expired connection cannot be leased to the requester, the pool may accumulate stale connections over time especially after a period of inactivity. It is generally advisable to force eviction of expired and idle connections from the pool after an extensive period of inactivity.

```
```

```
BasicConnPool connpool = <...>
connpool.closeExpired();
connpool.closeIdle(1, TimeUnit.MINUTES);
```

Generally it is considered to be a responsibility of the consumer to keep track of connections leased from the pool and to ensure their immediate release as soon as they are no longer needed or actively used. Nevertheless `BasicConnPool` provides protected methods to enumerate available idle connections and those currently leased from the pool. This enables the pool consumer to query connection state and selectively terminate connections meeting a particular criterion.

```
static class MyBasicConnPool extends BasicConnPool {

    @Override
    protected void enumAvailable(final PoolEntryCallback<HttpHost, HttpClientConnection> callback) {
        super.enumAvailable(callback);
    }

    @Override
    protected void enumLeased(final PoolEntryCallback<HttpHost, HttpClientConnection> callback) {
        super.enumLeased(callback);
    }

}
```

```
MyBasicConnPool connpool = new MyBasicConnPool();
connpool.enumAvailable(new PoolEntryCallback<HttpHost, HttpClientConnection>() {

    @Override
    public void process(final PoolEntry<HttpHost, HttpClientConnection> entry) {
        Date creationTime = new Date(entry.getCreated());
        if (creationTime.before(someTime)) {
            entry.close();
        }
    }

});
```

2.5. TLS/SSL support

Blocking connections can be bound to any arbitrary socket. This makes SSL support quite straightforward. Any `SSLSocket` instance can be bound to a blocking connection in order to make all messages transmitted over than connection secured by TLS/SSL.

```
SSLContext sslcontext = SSLContexts.createSystemDefault();
SocketFactory sf = sslcontext.getSocketFactory();
SSLSocket socket = (SSLSocket) sf.createSocket("somehost", 443);
// Enforce TLS and disable SSL
socket.setEnabledProtocols(new String[] {
        "TLSv1",
        "TLSv1.1",
        "TLSv1.2" });
// Enforce strong ciphers
socket.setEnabledCipherSuites(new String[] {
        "TLS_RSA_WITH_AES_256_CBC_SHA",
        "TLS_DHE_RSA_WITH_AES_256_CBC_SHA",
        "TLS_DHE_DSS_WITH_AES_256_CBC_SHA" });
DefaultBHttpClientConnection conn = new DefaultBHttpClientConnection(8 * 1204);
conn.bind(socket);
```

2.6. Embedded HTTP server

As of version 4.4 HttpCore ships with an embedded HTTP server based on blocking I/O components described above.

```
HttpRequestHandler requestHandler = <...>
HttpProcessor httpProcessor = <...>
SocketConfig socketConfig = SocketConfig.custom()
        .setSoTimeout(15000)
        .setTcpNoDelay(true)
        .build();
final HttpServer server = ServerBootstrap.bootstrap()
        .setListenerPort(8080)
        .setHttpProcessor(httpProcessor)
        .setSocketConfig(socketConfig)
        .setExceptionLogger(new StdErrorExceptionLogger())
        .registerHandler("*", requestHandler)
        .create();
server.start();
server.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);

Runtime.getRuntime().addShutdownHook(new Thread() {
    @Override
    public void run() {
        server.shutdown(5, TimeUnit.SECONDS);
    }
});
```

Chapter 3. Asynchronous I/O based on NIO

Asynchronous I/O model may be more appropriate for those scenarios where raw data throughput is less important than the ability to handle thousands of simultaneous connections in a scalable, resource efficient manner. Asynchronous I/O is arguably more complex and usually requires a special care when dealing with large message payloads.

3.1. Differences from other I/O frameworks

Solves similar problems as other frameworks, but has certain distinct features:

- minimalistic, optimized for data volume intensive protocols such as HTTP.
- efficient memory management: data consumer can read is only as much input data as it can process without having to allocate more memory.
- direct access to the NIO channels where possible.

3.2. I/O reactor

HttpCore NIO is based on the Reactor pattern as described by Doug Lea. The purpose of I/O reactors is to react to I/O events and to dispatch event notifications to individual I/O sessions. The main idea of I/O reactor pattern is to break away from the one thread per connection model imposed by the classic blocking I/O model. The `IOReactor` interface represents an abstract object which implements the Reactor pattern. Internally, `IOReactor` implementations encapsulate functionality of the NIO `java.nio.channels.Selector`.

I/O reactors usually employ a small number of dispatch threads (often as few as one) to dispatch I/O event notifications to a much greater number (often as many as several thousands) of I/O sessions or connections. It is generally recommended to have one dispatch thread per CPU core.

```
IOReactorConfig config = IOReactorConfig.DEFAULT;
IOReactor ioreactor = new DefaultConnectingIOReactor(config);
```

3.2.1. I/O dispatchers

`IOReactor` implementations make use of the `IOEventDispatch` interface to notify clients of events pending for a particular session. All methods of the `IOEventDispatch` are executed on a dispatch thread of the I/O reactor. Therefore, it is important that processing that takes place in the event methods will not block the dispatch thread for too long, as the I/O reactor will be unable to react to other events.

```
IOReactor ioreactor = new DefaultConnectingIOReactor();

IOEventDispatch eventDispatch = <...>
ioreactor.execute(eventDispatch);
```

Generic I/O events as defined by the `IOEventDispatch` interface:

- **`connected:`** Triggered when a new session has been created.
- **`inputReady:`** Triggered when the session has pending input.
- **`outputReady:`** Triggered when the session is ready for output.
- **`timeout:`** Triggered when the session has timed out.
- **`disconnected:`** Triggered when the session has been terminated.

3.2.2. I/O reactor shutdown

The shutdown of I/O reactors is a complex process and may usually take a while to complete. I/O reactors will attempt to gracefully terminate all active I/O sessions and dispatch threads approximately within the specified grace period. If any of the I/O sessions fails to terminate correctly, the I/O reactor will forcibly shut down remaining sessions.

```
IOReactor ioreactor = <...>
long gracePeriod = 3000L; // milliseconds
ioreactor.shutdown(gracePeriod);
```

The `IOReactor#shutdown(long)` method is safe to call from any thread.

3.2.3. I/O sessions

The `IOSession` interface represents a sequence of logically related data exchanges between two end points. `IOSession` encapsulates functionality of NIO `java.nio.channels.SelectionKey` and `java.nio.channels.SocketChannel`. The channel associated with the `IOSession` can be used to read data from and write data to the session.

```
IOSession iosession = <...>
ReadableByteChannel ch = (ReadableByteChannel) iosession.channel();
ByteBuffer dst = ByteBuffer.allocate(2048);
ch.read(dst);
```

3.2.4. I/O session state management

I/O sessions are not bound to an execution thread, therefore one cannot use the context of the thread to store a session's state. All details about a particular session must be stored within the session itself.

```
IOSession iosession = <...>
Object someState = <...>
iosession.setAttribute("state", someState);
...
IOSession iosession = <...>
Object currentState = iosession.getAttribute("state");
```

Please note that if several sessions make use of shared objects, access to those objects must be made thread-safe.

3.2.5. I/O session event mask

One can declare an interest in a particular type of I/O events for a particular I/O session by setting its event mask.

```
IOSession iosession = <...>
iosession.setEventMask(SelectionKey.OP_READ | SelectionKey.OP_WRITE);
```

One can also toggle `OP_READ` and `OP_WRITE` flags individually.

```
IOSession iosession = <...>
iosession.setEvent(SelectionKey.OP_READ);
iosession.clearEvent(SelectionKey.OP_READ);
```

Event notifications will not take place if the corresponding interest flag is not set.

3.2.6. I/O session buffers

Quite often I/O sessions need to maintain internal I/O buffers in order to transform input / output data prior to returning it to the consumer or writing it to the underlying channel. Memory management in HttpCore NIO is based on the fundamental principle that the data a consumer can read, is only as much input data as it can process without having to allocate more memory. That means, quite often some input data may remain unread in one of the internal or external session buffers. The I/O reactor can query the status of these session buffers, and make sure the consumer gets notified correctly as more data gets stored in one of the session buffers, thus allowing the consumer to read the remaining data once it is able to process it. I/O sessions can be made aware of the status of external session buffers using the `SessionBufferStatus` interface.

```
IOSession iosession = <...>
SessionBufferStatus myBufferStatus = <...>
iosession.setBufferStatus(myBufferStatus);
iosession.hasBufferedInput();
iosession.hasBufferedOutput();
```

3.2.7. I/O session shutdown

One can close an I/O session gracefully by calling `IOSession#close()` allowing the session to be closed in an orderly manner or by calling `IOSession#shutdown()` to forcibly close the underlying channel. The distinction between two methods is of primary importance for those types of I/O sessions that involve some sort of a session termination handshake such as SSL/TLS connections.

3.2.8. Listening I/O reactors

`ListeningIOReactor` represents an I/O reactor capable of listening for incoming connections on one or several ports.

```
ListeningIOReactor ioreactor = <...>
ListenerEndpoint ep1 = ioreactor.listen(new InetSocketAddress(8081) );
ListenerEndpoint ep2 = ioreactor.listen(new InetSocketAddress(8082));
ListenerEndpoint ep3 = ioreactor.listen(new InetSocketAddress(8083));
// Wait until all endpoints are up
ep1.waitFor();
ep2.waitFor();
ep3.waitFor();
```

Once an endpoint is fully initialized it starts accepting incoming connections and propagates I/O activity notifications to the `IOEventDispatch` instance.

One can obtain a set of registered endpoints at runtime, query the status of an endpoint at runtime, and close it if desired.

```
ListeningIOReactor ioreactor = <...>

Set<ListenerEndpoint> eps = ioreactor.getEndpoints();
for (ListenerEndpoint ep: eps) {
    // Still active?
    System.out.println(ep.getAddress());
    if (ep.isClosed()) {
        // If not, has it terminated due to an exception?
        if (ep.getException() != null) {
            ep.getException().printStackTrace();
        }
    } else {
        ep.close();
    }
}
```

3.2.9. Connecting I/O reactors

`ConnectingIOReactor` represents an I/O reactor capable of establishing connections with remote hosts.

```
ConnectingIOReactor ioreactor = <...>

SessionRequest sessionRequest = ioreactor.connect(
        new InetSocketAddress("www.google.com", 80),
        null, null, null);
```

Opening a connection to a remote host usually tends to be a time consuming process and may take a while to complete. One can monitor and control the process of session initialization by means of the `SessionRequest`interface.

```
// Make sure the request times out if connection
// has not been established after 1 sec
sessionRequest.setConnectTimeout(1000);
// Wait for the request to complete
sessionRequest.waitFor();
// Has request terminated due to an exception?
if (sessionRequest.getException() != null) {
    sessionRequest.getException().printStackTrace();
}
// Get hold of the new I/O session
IOSession iosession = sessionRequest.getSession();
```

`SessionRequest` implementations are expected to be thread-safe. Session request can be aborted at any time by calling `IOSession#cancel()` from another thread of execution.

```
if (!sessionRequest.isCompleted()) {
    sessionRequest.cancel();
}
```

One can pass several optional parameters to the `ConnectingIOReactor#connect()` method to exert a greater control over the process of session initialization.

A non-null local socket address parameter can be used to bind the socket to a specific local address.

```
ConnectingIOReactor ioreactor = <...>

SessionRequest sessionRequest = ioreactor.connect(
        new InetSocketAddress("www.google.com", 80),
        new InetSocketAddress("192.168.0.10", 1234),
        null, null);
```

One can provide an attachment object, which will be added to the new session's context upon initialization. This object can be used to pass an initial processing state to the protocol handler.

```
SessionRequest sessionRequest = ioreactor.connect(
        new InetSocketAddress("www.google.com", 80),
        null, new HttpHost("www.google.ru"), null);

IOSession iosession = sessionRequest.getSession();
HttpHost virtualHost = (HttpHost) iosession.getAttribute(
    IOSession.ATTACHMENT_KEY);
```

It is often desirable to be able to react to the completion of a session request asynchronously without having to wait for it, blocking the current thread of execution. One can optionally provide an implementation `SessionRequestCallback` interface to get notified of events related to session requests, such as request completion, cancellation, failure or timeout.

```
ConnectingIOReactor ioreactor = <...>

SessionRequest sessionRequest = ioreactor.connect(
        new InetSocketAddress("www.google.com", 80), null, null,
        new SessionRequestCallback() {

            public void cancelled(SessionRequest request) {
            }

            public void completed(SessionRequest request) {
                System.out.println("new connection to " +
                    request.getRemoteAddress());
            }

            public void failed(SessionRequest request) {
                if (request.getException() != null) {
                    request.getException().printStackTrace();
                }
            }

            public void timeout(SessionRequest request) {
            }

        });
```

3.3. I/O reactor configuration

I/O reactors by default use system dependent configuration which in most cases should be sensible enough.

```
IOReactorConfig config = IOReactorConfig.DEFAULT;
IOReactor ioreactor = new DefaultListeningIOReactor(config);
```

However in some cases custom settings may be necessary, for instance, in order to alter default socket properties and timeout values. One should rarely need to change other parameters.

```
IOReactorConfig config = IOReactorConfig.custom()
        .setTcpNoDelay(true)
        .setSoTimeout(5000)
        .setSoReuseAddress(true)
        .setConnectTimeout(5000)
        .build();
IOReactor ioreactor = new DefaultListeningIOReactor(config);
```

3.3.1. Queuing of I/O interest set operations

Several older JRE implementations (primarily from IBM) include what Java API documentation refers to as a naive implementation of the `java.nio.channels.SelectionKey` class. The problem with `java.nio.channels.SelectionKey` in such JREs is that reading or writing of the I/O interest set may block indefinitely if the I/O selector is in the process of executing a select operation. HttpCore NIO can be configured to operate in a special mode wherein I/O interest set operations are queued and executed by on the dispatch thread only when the I/O selector is not engaged in a select operation.

```
IOReactorConfig config = IOReactorConfig.custom()
        .setInterestOpQueued(true)
        .build();
```

3.4. I/O reactor exception handling

Protocol specific exceptions as well as those I/O exceptions thrown in the course of interaction with the session's channel are to be expected and are to be dealt with by specific protocol handlers. These exceptions may result in termination of an individual session but should not affect the I/O reactor and all other active sessions. There are situations, however, when the I/O reactor itself encounters an internal problem such as an I/O exception in the underlying NIO classes or an unhandled runtime exception. Those types of exceptions are usually fatal and will cause the I/O reactor to shut down automatically.

There is a possibility to override this behavior and prevent I/O reactors from shutting down automatically in case of a runtime exception or an I/O exception in internal classes. This can be accomplished by providing a custom implementation of the `IOReactorExceptionHandler` interface.

```
DefaultConnectingIOReactor ioreactor = <...>

ioreactor.setExceptionHandler(new IOReactorExceptionHandler() {

    public boolean handle(IOException ex) {
        if (ex instanceof BindException) {
            // bind failures considered OK to ignore
            return true;
        }
        return false;
    }

    public boolean handle(RuntimeException ex) {
        if (ex instanceof UnsupportedOperationException) {
            // Unsupported operations considered OK to ignore
            return true;
        }
        return false;
    }

});
```

One needs to be very careful about discarding exceptions indiscriminately. It is often much better to let the I/O reactor shut down itself cleanly and restart it rather than leaving it in an inconsistent or unstable state.

3.4.1. I/O reactor audit log

If an I/O reactor is unable to automatically recover from an I/O or a runtime exception it will enter the shutdown mode. First off, it will close all active listeners and cancel all pending new session requests. Then it will attempt to close all active I/O sessions gracefully giving them some time to flush pending output data and terminate cleanly. Lastly, it will forcibly shut down those I/O sessions that still remain active after the grace period. This is a fairly complex process, where many things can fail at the same time and many different exceptions can be thrown in the course of the shutdown process. The I/O reactor will record all exceptions thrown during the shutdown process, including the original one that actually caused the shutdown in the first place, in an audit log. One can examine the audit log and decide whether it is safe to restart the I/O reactor.

```
DefaultConnectingIOReactor ioreactor = <...>

// Give it 5 sec grace period
ioreactor.shutdown(5000);
List<ExceptionEvent> events = ioreactor.getAuditLog();
for (ExceptionEvent event: events) {
    System.err.println("Time: " + event.getTimestamp());
    event.getCause().printStackTrace();
}
```

3.5. Non-blocking HTTP connections

Effectively non-blocking HTTP connections are wrappers around `IOSession` with HTTP specific functionality. Non-blocking HTTP connections are stateful and not thread-safe. Input / output operations on non-blocking HTTP connections should be restricted to the dispatch events triggered by the I/O event dispatch thread.

3.5.1. Execution context of non-blocking HTTP connections

Non-blocking HTTP connections are not bound to a particular thread of execution and therefore they need to maintain their own execution context. Each non-blocking HTTP connection has an `HttpContext` instance associated with it, which can be used to maintain a processing state. The `HttpContext` instance is thread-safe and can be manipulated from multiple threads.

```
DefaultNHttpClientConnection conn = <...>
Object myStateObject = <...>

HttpContext context = conn.getContext();
context.setAttribute("state", myStateObject);
```

3.5.2. Working with non-blocking HTTP connections

At any point of time one can obtain the request and response objects currently being transferred over the non-blocking HTTP connection. Any of these objects, or both, can be null if there is no incoming or outgoing message currently being transferred.

```
NHttpConnection conn = <...>
```

```
HttpRequest request = conn.getHttpRequest();
if (request != null) {
    System.out.println("Transferring request: " +
            request.getRequestLine());
}
HttpResponse response = conn.getHttpResponse();
if (response != null) {
    System.out.println("Transferring response: " +
            response.getStatusLine());
}
```

However, please note that the current request and the current response may not necessarily represent the same message exchange! Non-blocking HTTP connections can operate in a full duplex mode. One can process incoming and outgoing messages completely independently from one another. This makes non-blocking HTTP connections fully pipelining capable, but at same time implies that this is the job of the protocol handler to match logically related request and the response messages.

Over-simplified process of submitting a request on the client side may look like this:

```
NHttpClientConnection conn = <...>
// Obtain execution context
HttpContext context = conn.getContext();
// Obtain processing state
Object state = context.getAttribute("state");
// Generate a request based on the state information
HttpRequest request = new BasicHttpRequest("GET", "/");

conn.submitRequest(request);
System.out.println(conn.isRequestSubmitted());
```

Over-simplified process of submitting a response on the server side may look like this:

```
NHttpServerConnection conn = <...>
// Obtain execution context
HttpContext context = conn.getContext();
// Obtain processing state
Object state = context.getAttribute("state");

// Generate a response based on the state information
HttpResponse response = new BasicHttpResponse(HttpVersion.HTTP_1_1,
    HttpStatus.SC_OK, "OK");
BasicHttpEntity entity = new BasicHttpEntity();
entity.setContentType("text/plain");
entity.setChunked(true);
response.setEntity(entity);

conn.submitResponse(response);
System.out.println(conn.isResponseSubmitted());
```

Please note that one should rarely need to transmit messages using these low level methods and should use appropriate higher level HTTP service implementations instead.

3.5.3. HTTP I/O control

All non-blocking HTTP connections classes implement `IOControl` interface, which represents a subset of connection functionality for controlling interest in I/O even notifications. `IOControl` instances are expected to be fully thread-safe. Therefore `IOControl` can be used to request / suspend I/O event notifications from any thread.

One must take special precautions when interacting with non-blocking connections. `HttpRequest` and `HttpResponse` are not thread-safe. It is generally advisable that all input / output operations on a non-blocking connection are executed from the I/O event dispatch thread.

The following pattern is recommended:

- Use `IOControl` interface to pass control over connection's I/O events to another thread / session.
- If input / output operations need be executed on that particular connection, store all the required information (state) in the connection context and request the appropriate I/O operation by calling `IOControl#requestInput()` or `IOControl#requestOutput()` method.
- Execute the required operations from the event method on the dispatch thread using information stored in connection context.

Please note all operations that take place in the event methods should not block for too long, because while the dispatch thread remains blocked in one session, it is unable to process events for all other sessions. I/O operations with the underlying channel of the session are not a problem as they are guaranteed to be non-blocking.

3.5.4. Non-blocking content transfer

The process of content transfer for non-blocking connections works completely differently compared to that of blocking connections, as non-blocking connections need to accommodate to the asynchronous nature of the NIO model. The main distinction between two types of connections is inability to use the usual, but inherently blocking `java.io.InputStream` and `java.io.OutputStream` classes to represent streams of inbound and outbound content. HttpCore NIO provides `ContentEncoder` and `ContentDecoder` interfaces to handle the process of asynchronous content transfer. Non-blocking HTTP connections will instantiate the appropriate implementation of a content codec based on properties of the entity enclosed with the message.

Non-blocking HTTP connections will fire input events until the content entity is fully transferred.

```
ContentDecoder decoder = <...>
//Read data in
ByteBuffer dst = ByteBuffer.allocate(2048);
decoder.read(dst);
// Decode will be marked as complete when
// the content entity is fully transferred
if (decoder.isCompleted()) {
    // Done
}
```

Non-blocking HTTP connections will fire output events until the content entity is marked as fully transferred.

```
ContentEncoder encoder = <...>
// Prepare output data
ByteBuffer src = ByteBuffer.allocate(2048);
// Write data out
encoder.write(src);
// Mark content entity as fully transferred when done
encoder.complete();
```

Please note, one still has to provide an HttpEntity instance when submitting an entity enclosing message to the non-blocking HTTP connection. Properties of that entity will be used to initialize an `ContentEncoder` instance to be used for transferring entity content. Non-blocking HTTP connections, however, ignore inherently blocking `HttpEntity#getContent()` and `HttpEntity#writeTo()` methods of the enclosed entities.

```
NHttpServerConnection conn  = <...>

HttpResponse response = new BasicHttpResponse(HttpVersion.HTTP_1_1,
    HttpStatus.SC_OK, "OK");
BasicHttpEntity entity = new BasicHttpEntity();
entity.setContentType("text/plain");
entity.setChunked(true);
entity.setContent(null);
response.setEntity(entity);

conn.submitResponse(response);
```

Likewise, incoming entity enclosing message will have an `HttpEntity` instance associated with them, but an attempt to call `HttpEntity#getContent()` or `HttpEntity#writeTo()` methods will cause an `java.lang.IllegalStateException`. The `HttpEntity` instance can be used to determine properties of the incoming entity such as content length.

```
NHttpClientConnection conn = <...>

HttpResponse response = conn.getHttpResponse();
HttpEntity entity = response.getEntity();
if (entity != null) {
    System.out.println(entity.getContentType());
    System.out.println(entity.getContentLength());
    System.out.println(entity.isChunked());
}
```

3.5.5. Supported non-blocking content transfer mechanisms

Default implementations of the non-blocking HTTP connection interfaces support three content transfer mechanisms defined by the HTTP/1.1 specification:

- **`Content-Length` delimited:** The end of the content entity is determined by the value of the `Content-Length` header. Maximum entity length: `Long#MAX_VALUE`.
- **Identity coding:** The end of the content entity is demarcated by closing the underlying connection (end of stream condition). For obvious reasons the identity encoding can only be used on the server side. Max entity length: unlimited.
- **Chunk coding:** The content is sent in small chunks. Max entity length: unlimited.

The appropriate content codec will be created automatically depending on properties of the entity enclosed with the message.

3.5.6. Direct channel I/O

Content codes are optimized to read data directly from or write data directly to the underlying I/O session's channel, whenever possible avoiding intermediate buffering in a session buffer. Moreover,

those codecs that do not perform any content transformation (`Content-Length` delimited and identity codecs, for example) can leverage NIO `java.nio.FileChannel` methods for significantly improved performance of file transfer operations both inbound and outbound.

If the actual content decoder implements `FileContentDecoder` one can make use of its methods to read incoming content directly to a file bypassing an intermediate `java.nio.ByteBuffer`.

```
ContentDecoder decoder = <...>
//Prepare file channel
FileChannel dst;
//Make use of direct file I/O if possible
if (decoder instanceof FileContentDecoder) {
    long Bytesread = ((FileContentDecoder) decoder)
        .transfer(dst, 0, 2048);
     // Decode will be marked as complete when
     // the content entity is fully transmitted
     if (decoder.isCompleted()) {
         // Done
     }
}
```

If the actual content encoder implements `FileContentEncoder` one can make use of its methods to write outgoing content directly from a file bypassing an intermediate `java.nio.ByteBuffer`.

```
ContentEncoder encoder = <...>
// Prepare file channel
FileChannel src;
// Make use of direct file I/O if possible
if (encoder instanceof FileContentEncoder) {
    // Write data out
    long bytesWritten = ((FileContentEncoder) encoder)
        .transfer(src, 0, 2048);
    // Mark content entity as fully transferred when done
    encoder.complete();
}
```

3.6. HTTP I/O event dispatchers

HTTP I/O event dispatchers serve to convert generic I/O events triggered by an I/O reactor to HTTP protocol specific events. They rely on `NHttpClientEventHandler` and `NHttpServerEventHandler` interfaces to propagate HTTP protocol events to a HTTP protocol handler.

Server side HTTP I/O events as defined by the `NHttpServerEventHandler` interface:

- **`connected`:** Triggered when a new incoming connection has been created.
- **`requestReceived`:** Triggered when a new HTTP request is received. The connection passed as a parameter to this method is guaranteed to return a valid HTTP request object. If the request received encloses a request entity this method will be followed a series of `inputReady` events to transfer the request content.
- **`inputReady`:** Triggered when the underlying channel is ready for reading a new portion of the request entity through the corresponding content decoder. If the content consumer is unable to process the incoming content, input event notifications can temporarily suspended using `IOControl` interface (super interface of `NHttpServerConnection`). Please note that the

`NHttpServerConnection` and `ContentDecoder` objects are not thread-safe and should only be used within the context of this method call. The `IOControl` object can be shared and used on other thread to resume input event notifications when the handler is capable of processing more content.

- **`responseReady:`** Triggered when the connection is ready to accept new HTTP response. The protocol handler does not have to submit a response if it is not ready.

- **`outputReady:`** Triggered when the underlying channel is ready for writing a next portion of the response entity through the corresponding content encoder. If the content producer is unable to generate the outgoing content, output event notifications can be temporarily suspended using `IOControl` interface (super interface of `NHttpServerConnection`). Please note that the `NHttpServerConnection` and `ContentEncoder` objects are not thread-safe and should only be used within the context of this method call. The `IOControl` object can be shared and used on other thread to resume output event notifications when more content is made available.

- **`exception:`** Triggered when an I/O error occurrs while reading from or writing to the underlying channel or when an HTTP protocol violation occurs while receiving an HTTP request.

- **`timeout:`** Triggered when no input is detected on this connection over the maximum period of inactivity.

- **`closed:`** Triggered when the connection has been closed.

Client side HTTP I/O events as defined by the `NHttpClientEventHandler` interface:

- **`connected:`** Triggered when a new outgoing connection has been created. The attachment object passed as a parameter to this event is an arbitrary object that was attached to the session request.

- **`requestReady:`** Triggered when the connection is ready to accept new HTTP request. The protocol handler does not have to submit a request if it is not ready.

- **`outputReady:`** Triggered when the underlying channel is ready for writing a next portion of the request entity through the corresponding content encoder. If the content producer is unable to generate the outgoing content, output event notifications can be temporarily suspended using `IOControl` interface (super interface of `NHttpClientConnection`). Please note that the `NHttpClientConnection` and `ContentEncoder` objects are not thread-safe and should only be used within the context of this method call. The `IOControl` object can be shared and used on other thread to resume output event notifications when more content is made available.

- **`responseReceived:`** Triggered when an HTTP response is received. The connection passed as a parameter to this method is guaranteed to return a valid HTTP response object. If the response received encloses a response entity this method will be followed a series of `inputReady` events to transfer the response content.

- **`inputReady:`** Triggered when the underlying channel is ready for reading a new portion of the response entity through the corresponding content decoder. If the content consumer is unable to process the incoming content, input event notifications can be temporarily suspended using `IOControl` interface (super interface of `NHttpClientConnection`). Please note that the `NHttpClientConnection` and `ContentDecoder` objects are not thread-safe and should only be used within the context of this method call. The `IOControl` object can be shared and used on other thread to resume input event notifications when the handler is capable of processing more content.

- **`exception:`** Triggered when an I/O error occurs while reading from or writing to the underlying channel or when an HTTP protocol violation occurs while receiving an HTTP response.

- **`timeout:`** Triggered when no input is detected on this connection over the maximum period of inactivity.

- **`closed:`** Triggered when the connection has been closed.

3.7. Non-blocking HTTP content producers

As discussed previously the process of content transfer for non-blocking connections works completely differently compared to that for blocking connections. For obvious reasons classic I/O abstraction based on inherently blocking `java.io.InputStream` and `java.io.OutputStream` classes is not well suited for asynchronous data transfer. In order to avoid inefficient and potentially blocking I/O operation redirection through `java.nio.channels.Channles#newChannel` non-blocking HTTP entities are expected to implement NIO specific extension interface `HttpAsyncContentProducer`.

The `HttpAsyncContentProducer` interface defines several additional method for efficient streaming of content to a non-blocking HTTP connection:

- **`produceContent:`** Invoked to write out a chunk of content to the `ContentEncoder` . The `IOControl` interface can be used to suspend output events if the entity is temporarily unable to produce more content. When all content is finished, the producer MUST call `ContentEncoder#complete()`. Failure to do so may cause the entity to be incorrectly delimited. Please note that the `ContentEncoder` object is not thread-safe and should only be used within the context of this method call. The `IOControl` object can be shared and used on other thread resume output event notifications when more content is made available.

- **`isRepeatable:`** Determines whether or not this producer is capable of producing its content more than once. Repeatable content producers are expected to be able to recreate their content even after having been closed.

- **`close:`** Closes the producer and releases all resources currently allocated by it.

3.7.1. Creating non-blocking entities

Several HTTP entity implementations included in HttpCore NIO support `HttpAsyncContentProducer` interface:

- `NByteArrayEntity`

- `NStringEntity`

- `NFileEntity`

3.7.1.1. `NByteArrayEntity`

This is a simple self-contained repeatable entity, which receives its content from a given byte array. This byte array is supplied to the constructor.

```
NByteArrayEntity entity = new NByteArrayEntity(new byte[] {1, 2, 3});
```

3.7.1.2. `NStringEntity`

This is a simple, self-contained, repeatable entity that retrieves its data from a `java.lang.String` object. It has 2 constructors, one simply constructs with a given string where the other also takes a character encoding for the data in the `java.lang.String`.

```
NStringEntity myEntity = new NStringEntity("important message",
        Consts.UTF_8);
```

3.7.1.3. `NFileEntity`

This entity reads its content body from a file. This class is mostly used to stream large files of different types, so one needs to supply the content type of the file to make sure the content can be correctly recognized and processed by the recipient.

```
File staticFile = new File("/path/to/myapp.jar");
NFileEntity entity = new NFileEntity(staticFile,
    ContentType.create("application/java-archive", null));
```

The `NHttpEntity` will make use of the direct channel I/O whenever possible, provided the content encoder is capable of transferring data directly from a file to the socket of the underlying connection.

3.8. Non-blocking HTTP protocol handlers

3.8.1. Asynchronous HTTP service

`HttpAsyncService` is a fully asynchronous HTTP server side protocol handler based on the non-blocking (NIO) I/O model. `HttpAsyncService` translates individual events fired through the `NHttpServerEventHandler` interface into logically related HTTP message exchanges.

Upon receiving an incoming request the `HttpAsyncService` verifies the message for compliance with the server expectations using `HttpAsyncExpectationVerifier`, if provided, and then `HttpAsyncRequestHandlerResolver` is used to resolve the request URI to a particular `HttpAsyncRequestHandler` intended to handle the request with the given URI. The protocol handler uses the selected `HttpAsyncRequestHandler` instance to process the incoming request and to generate an outgoing response.

`HttpAsyncService` relies on `HttpProcessor` to generate mandatory protocol headers for all outgoing messages and apply common, cross-cutting message transformations to all incoming and outgoing messages, whereas individual HTTP request handlers are expected to implement application specific content generation and processing.

```
HttpProcessor httpproc = HttpProcessorBuilder.create()
        .add(new ResponseDate())
        .add(new ResponseServer("MyServer-HTTP/1.1"))
        .add(new ResponseContent())
        .add(new ResponseConnControl())
        .build();
HttpAsyncService protocolHandler = new HttpAsyncService(httpproc, null);
IOEventDispatch ioEventDispatch = new DefaultHttpServerIODispatch(
        protocolHandler,
        new DefaultNHttpServerConnectionFactory(ConnectionConfig.DEFAULT));
```

```
ListeningIOReactor ioreactor = new DefaultListeningIOReactor();
ioreactor.execute(ioEventDispatch);
```

3.8.1.1. Non-blocking HTTP request handlers

`HttpAsyncRequestHandler` represents a routine for asynchronous processing of a specific group of non-blocking HTTP requests. Protocol handlers are designed to take care of protocol specific aspects, whereas individual request handlers are expected to take care of application specific HTTP processing. The main purpose of a request handler is to generate a response object with a content entity to be sent back to the client in response to the given request.

```
HttpAsyncRequestHandler<HttpRequest> rh = new HttpAsyncRequestHandler<HttpRequest>() {

    public HttpAsyncRequestConsumer<HttpRequest> processRequest(
            final HttpRequest request,
            final HttpContext context) {
        // Buffer request content in memory for simplicity
        return new BasicAsyncRequestConsumer();
    }

    public void handle(
            final HttpRequest request,
            final HttpAsyncExchange httpexchange,
            final HttpContext context) throws HttpException, IOException {
        HttpResponse response = httpexchange.getResponse();
        response.setStatusCode(HttpStatus.SC_OK);
        NFileEntity body = new NFileEntity(new File("static.html"),
                ContentType.create("text/html", Consts.UTF_8));
        response.setEntity(body);
        httpexchange.submitResponse(new BasicAsyncResponseProducer(response));
    }

};
```

Request handlers must be implemented in a thread-safe manner. Similarly to servlets, request handlers should not use instance variables unless access to those variables are synchronized.

3.8.1.2. Asynchronous HTTP exchange

The most fundamental difference of the non-blocking request handlers compared to their blocking counterparts is ability to defer transmission of the HTTP response back to the client without blocking the I/O thread by delegating the process of handling the HTTP request to a worker thread or another service. The instance of `HttpAsyncExchange` passed as a parameter to the `HttpAsyncRequestHandler#handle` method to submit a response as at a later point once response content becomes available.

The `HttpAsyncExchange` interface can be interacted with using the following methods:

- **`getRequest:`** Returns the received HTTP request message.
- **`getResponse:`** Returns the default HTTP response message that can submitted once ready.
- **`submitResponse:`** Submits an HTTP response and completed the message exchange.
- **`isCompleted:`** Determines whether or not the message exchange has been completed.
- **`setCallback:`** Sets `Cancellable` callback to be invoked in case the underlying connection times out or gets terminated prematurely by the client. This callback can be used to cancel a long running response generating process if a response is no longer needed.

- **`setTimeout:`** Sets timeout for this message exchange.
- **`getTimeout:`** Returns timeout for this message exchange.

```
HttpAsyncRequestHandler<HttpRequest> rh = new HttpAsyncRequestHandler<HttpRequest>() {

    public HttpAsyncRequestConsumer<HttpRequest> processRequest(
            final HttpRequest request,
            final HttpContext context) {
        // Buffer request content in memory for simplicity
        return new BasicAsyncRequestConsumer();
    }

    public void handle(
            final HttpRequest request,
            final HttpAsyncExchange httpexchange,
            final HttpContext context) throws HttpException, IOException {

        new Thread() {

            @Override
            public void run() {
                try {
                    Thread.sleep(10);
                }
                catch(InterruptedException ie) {}
                HttpResponse response = httpexchange.getResponse();
                response.setStatusCode(HttpStatus.SC_OK);
                NFileEntity body = new NFileEntity(new File("static.html"),
                        ContentType.create("text/html", Consts.UTF_8));
                response.setEntity(body);
                httpexchange.submitResponse(new BasicAsyncResponseProducer(response));
            }
        }.start();

    }

};
```

Please note `HttpResponse` instances are not thread-safe and may not be modified concurrently. Non-blocking request handlers must ensure HTTP response cannot be accessed by more than one thread at a time.

3.8.1.3. Asynchronous HTTP request consumer

`HttpAsyncRequestConsumer` facilitates the process of asynchronous processing of HTTP requests. It is a callback interface used by `HttpAsyncRequestHandlers` to process an incoming HTTP request message and to stream its content from a non-blocking server side HTTP connection.

HTTP I/O events and methods as defined by the `HttpAsyncRequestConsumer` interface:

- **`requestReceived:`** Invoked when a HTTP request message is received.
- **`consumeContent:`** Invoked to process a chunk of content from the `ContentDecoder`. The `IOControl` interface can be used to suspend input events if the consumer is temporarily unable to consume more content. The consumer can use the `ContentDecoder#isCompleted()` method to find out whether or not the message content has been fully consumed. Please note that the `ContentDecoder` object is not thread-safe and should only be used within the context of this method call. The `IOControl` object can be shared and used on other thread to resume input event notifications

when the consumer is capable of processing more content. This event is invoked only if the incoming request message has a content entity enclosed in it.

- **`requestCompleted:`** Invoked to signal that the request has been fully processed.
- **`failed:`** Invoked to signal that the request processing terminated abnormally.
- **`getException:`** Returns an exception in case of an abnormal termination. This method returns `null` if the request execution is still ongoing or if it completed successfully.
- **`getResult:`** Returns a result of the request execution, when available. This method returns `null` if the request execution is still ongoing.
- **`isDone:`** Determines whether or not the request execution completed. If the request processing terminated normally `getResult()` can be used to obtain the result. If the request processing terminated abnormally `getException()` can be used to obtain the cause.
- **`close:`** Closes the consumer and releases all resources currently allocated by it.

`HttpAsyncRequestConsumer` implementations are expected to be thread-safe.

`BasicAsyncRequestConsumer` is a very basic implementation of the `HttpAsyncRequestConsumer` interface shipped with the library. Please note that this consumer buffers request content in memory and therefore should be used for relatively small request messages.

3.8.1.4. Asynchronous HTTP response producer

`HttpAsyncResponseProducer` facilitates the process of asynchronous generation of HTTP responses. It is a callback interface used by `HttpAsyncRequestHandlers` to generate an HTTP response message and to stream its content to a non-blocking server side HTTP connection.

HTTP I/O events and methods as defined by the `HttpAsyncResponseProducer` interface:

- **`generateResponse:`** Invoked to generate a HTTP response message header.
- **`produceContent:`** Invoked to write out a chunk of content to the `ContentEncoder`. The `IOControl` interface can be used to suspend output events if the producer is temporarily unable to produce more content. When all content is finished, the producer MUST call `ContentEncoder#complete()`. Failure to do so may cause the entity to be incorrectly delimited. Please note that the `ContentEncoder` object is not thread-safe and should only be used within the context of this method call. The `IOControl` object can be shared and used on other thread resume output event notifications when more content is made available. This event is invoked only for if the outgoing response message has a content entity enclosed in it, that is `HttpResponse#getEntity()` returns `null`.
- **`responseCompleted:`** Invoked to signal that the response has been fully written out.
- **`failed:`** Invoked to signal that the response processing terminated abnormally.
- **`close:`** Closes the producer and releases all resources currently allocated by it.

`HttpAsyncResponseProducer` implementations are expected to be thread-safe.

`BasicAsyncResponseProducer` is a basic implementation of the `HttpAsyncResponseProducer` interface shipped with the library. The producer can make use of the `HttpAsyncContentProducer`

interface to efficiently stream out message content to a non-blocking HTTP connection, if it is implemented by the `HttpEntity` enclosed in the response.

3.8.1.5. Non-blocking request handler resolver

The management of non-blocking HTTP request handlers is quite similar to that of blocking HTTP request handlers. Usually an instance of `HttpAsyncRequestHandlerResolver` is used to maintain a registry of request handlers and to matches a request URI to a particular request handler. HttpCore includes only a very simple implementation of the request handler resolver based on a trivial pattern matching algorithm: `HttpAsyncRequestHandlerRegistry` supports only three formats: `*`, `<uri>*` and `*<uri>`.

```
HttpAsyncRequestHandler<?> myRequestHandler1 = <...>
HttpAsyncRequestHandler<?> myRequestHandler2 = <...>
HttpAsyncRequestHandler<?> myRequestHandler3 = <...>
UriHttpAsyncRequestHandlerMapper handlerReqistry =
        new UriHttpAsyncRequestHandlerMapper();
handlerReqistry.register("/service/*", myRequestHandler1);
handlerReqistry.register("*.do", myRequestHandler2);
handlerReqistry.register("*", myRequestHandler3);
```

Users are encouraged to provide more sophisticated implementations of `HttpAsyncRequestHandlerResolver`, for instance, based on regular expressions.

3.8.2. Asynchronous HTTP request executor

`HttpAsyncRequestExecutor` is a fully asynchronous client side HTTP protocol handler based on the NIO (non-blocking) I/O model. `HttpAsyncRequestExecutor` translates individual events fired through the `NHttpClientEventHandler` interface into logically related HTTP message exchanges.

`HttpAsyncRequestExecutor` relies on `HttpAsyncRequestExecutionHandler` to implement application specific content generation and processing and to handle logically related series of HTTP request / response exchanges, which may also span across multiple connections. `HttpProcessor` provided by the `HttpAsyncRequestExecutionHandler` instance will be used to generate mandatory protocol headers for all outgoing messages and apply common, cross-cutting message transformations to all incoming and outgoing messages. The caller is expected to pass an instance of `HttpAsyncRequestExecutionHandler` to be used for the next series of HTTP message exchanges through the connection context using `HttpAsyncRequestExecutor#HTTP_HANDLER` attribute. HTTP exchange sequence is considered complete when the `HttpAsyncRequestExecutionHandler#isDone()` method returns `true`.

```
HttpAsyncRequestExecutor ph = new HttpAsyncRequestExecutor();
IOEventDispatch ioEventDispatch = new DefaultHttpClientIODispatch(ph,
        new DefaultNHttpClientConnectionFactory(ConnectionConfig.DEFAULT));
ConnectingIOReactor ioreactor = new DefaultConnectingIOReactor();
ioreactor.execute(ioEventDispatch);
```

The `HttpAsyncRequester` utility class can be used to abstract away low level details of `HttpAsyncRequestExecutionHandler` management. Please note `HttpAsyncRequester` supports single HTTP request / response exchanges only. It does not support HTTP authentication and does not handle redirects automatically.

```
```

```
HttpProcessor httpproc = HttpProcessorBuilder.create()
        .add(new RequestContent())
        .add(new RequestTargetHost())
        .add(new RequestConnControl())
        .add(new RequestUserAgent("MyAgent-HTTP/1.1"))
        .add(new RequestExpectContinue(true))
        .build();
HttpAsyncRequester requester = new HttpAsyncRequester(httpproc);
NHttpClientConnection conn = <...>
Future<HttpResponse> future = requester.execute(
        new BasicAsyncRequestProducer(
                new HttpHost("localhost"),
                new BasicHttpRequest("GET", "/")),
        new BasicAsyncResponseConsumer(),
        conn);
HttpResponse response = future.get();
```

3.8.2.1. Asynchronous HTTP request producer

`HttpAsyncRequestProducer` facilitates the process of asynchronous generation of HTTP requests. It is a callback interface whose methods get invoked to generate an HTTP request message and to stream message content to a non-blocking client side HTTP connection.

Repeatable request producers capable of generating the same request message more than once can be reset to their initial state by calling the `resetRequest()` method, at which point request producers are expected to release currently allocated resources that are no longer needed or re-acquire resources needed to repeat the process.

HTTP I/O events and methods as defined by the `HttpAsyncRequestProducer` interface:

- **`getTarget`:** Invoked to obtain the request target host.
- **`generateRequest`:** Invoked to generate a HTTP request message header. The message is expected to implement the `HttpEntityEnclosingRequest` interface if it is to enclose a content entity.
- **`produceContent`:** Invoked to write out a chunk of content to the `ContentEncoder`. The `IOControl` interface can be used to suspend output events if the producer is temporarily unable to produce more content. When all content is finished, the producer MUST call `ContentEncoder#complete()`. Failure to do so may cause the entity to be incorrectly delimited Please note that the `ContentEncoder` object is not thread-safe and should only be used within the context of this method call. The `IOControl` object can be shared and used on other thread resume output event notifications when more content is made available. This event is invoked only for if the outgoing request message has a content entity enclosed in it, that is `HttpEntityEnclosingRequest#getEntity()` returns `null` .
- **`requestCompleted`:** Invoked to signal that the request has been fully written out.
- **`failed`:** Invoked to signal that the request processing terminated abnormally.
- **`resetRequest`:** Invoked to reset the producer to its initial state. Repeatable request producers are expected to release currently allocated resources that are no longer needed or re-acquire resources needed to repeat the process.
- **`close`:** Closes the producer and releases all resources currently allocated by it.

`HttpAsyncRequestProducer` implementations are expected to be thread-safe.

`BasicAsyncRequestProducer` is a basic implementation of the `HttpAsyncRequestProducer` interface shipped with the library. The producer can make use of the `HttpAsyncContentProducer` interface to efficiently stream out message content to a non-blocking HTTP connection, if it is implemented by the `HttpEntity` enclosed in the request.

3.8.2.2. Asynchronous HTTP response consumer

`HttpAsyncResponseConsumer` facilitates the process of asynchronous processing of HTTP responses. It is a callback interface whose methods get invoked to process an HTTP response message and to stream message content from a non-blocking client side HTTP connection.

HTTP I/O events and methods as defined by the `HttpAsyncResponseConsumer` interface:

- **`responseReceived`:** Invoked when a HTTP response message is received.
- **`consumeContent`:** Invoked to process a chunk of content from the `ContentDecoder`. The `IOControl` interface can be used to suspend input events if the consumer is temporarily unable to consume more content. The consumer can use the `ContentDecoder#isCompleted()` method to find out whether or not the message content has been fully consumed. Please note that the `ContentDecoder` object is not thread-safe and should only be used within the context of this method call. The `IOControl` object can be shared and used on other thread to resume input event notifications when the consumer is capable of processing more content. This event is invoked only for if the incoming response message has a content entity enclosed in it.
- **`responseCompleted`:** Invoked to signal that the response has been fully processed.
- **`failed`:** Invoked to signal that the response processing terminated abnormally.
- **`getException`:** Returns an exception in case of an abnormal termination. This method returns `null` if the response processing is still ongoing or if it completed successfully.
- **`getResult`:** Returns a result of the response processing, when available. This method returns `null` if the response processing is still ongoing.
- **`isDone`:** Determines whether or not the response processing completed. If the response processing terminated normally `getResult()` can be used to obtain the result. If the response processing terminated abnormally `getException()` can be used to obtain the cause.
- **`close`:** Closes the consumer and releases all resources currently allocated by it.

`HttpAsyncResponseConsumer` implementations are expected to be thread-safe.

`BasicAsyncResponseConsumer` is a very basic implementation of the `HttpAsyncResponseConsumer` interface shipped with the library. Please note that this consumer buffers response content in memory and therefore should be used for relatively small response messages.

3.9. Non-blocking connection pools

Non-blocking connection pools are quite similar to blocking one with one significant distinction that they have to reply an I/O reactor to establish new connections. As a result connections leased from a non-blocking pool are returned fully initialized and already bound to a particular I/O session. Non-blocking connections managed by a connection pool cannot be bound to an arbitrary I/O session.

```
HttpHost target = new HttpHost("localhost");
ConnectingIOReactor ioreactor = <...>
BasicNIOConnPool connpool = new BasicNIOConnPool(ioreactor);
connpool.lease(target, null,
        10, TimeUnit.SECONDS,
        new FutureCallback<BasicNIOPoolEntry>() {
            @Override
            public void completed(BasicNIOPoolEntry entry) {
                NHttpClientConnection conn = entry.getConnection();
                System.out.println("Connection successfully leased");
                // Update connection context and request output
                conn.requestOutput();
            }

            @Override
            public void failed(Exception ex) {
                System.out.println("Connection request failed");
                ex.printStackTrace();
            }

            @Override
            public void cancelled() {
            }
        });
```

Please note due to event-driven nature of asynchronous communication model it is quite difficult to ensure proper release of persistent connections back to the pool. One can make use of `HttpAsyncRequester` to handle connection lease and release behind the scene.

```
ConnectingIOReactor ioreactor = <...>
HttpProcessor httpproc = <...>
BasicNIOConnPool connpool = new BasicNIOConnPool(ioreactor);
HttpAsyncRequester requester = new HttpAsyncRequester(httpproc);
HttpHost target = new HttpHost("localhost");
Future<HttpResponse> future = requester.execute(
        new BasicAsyncRequestProducer(
                new HttpHost("localhost"),
                new BasicHttpRequest("GET", "/")),
        new BasicAsyncResponseConsumer(),
        connpool);
```

3.10. Pipelined request execution

In addition to the normal request / response execution mode `HttpAsyncRequester` is also capable of executing requests in the so called pipelined mode whereby several requests are immediately written out to the underlying connection. Please note that entity enclosing requests can be executed in the pipelined mode but the 'expect: continue' handshake should be disabled (request messages should contains no 'Expect: 100-continue' header).

```
HttpProcessor httpproc = <...>
HttpAsyncRequester requester = new HttpAsyncRequester(httpproc);
HttpHost target = new HttpHost("www.apache.org");
List<BasicAsyncRequestProducer> requestProducers = Arrays.asList(
        new BasicAsyncRequestProducer(target, new BasicHttpRequest("GET", "/index.html")),
        new BasicAsyncRequestProducer(target, new BasicHttpRequest("GET", "/foundation/index.html")),
        new BasicAsyncRequestProducer(target, new BasicHttpRequest("GET", "/foundation/how-it-works.ht
);
List<BasicAsyncResponseConsumer> responseConsumers = Arrays.asList(
        new BasicAsyncResponseConsumer(),
```

```
        new BasicAsyncResponseConsumer(),
        new BasicAsyncResponseConsumer()
);
HttpCoreContext context = HttpCoreContext.create();
Future<List<HttpResponse>> future = requester.executePipelined(
        target, requestProducers, responseConsumers, pool, context, null);
```

Please note that older web servers and especially older HTTP proxies may be unable to handle pipelined requests correctly. Use the pipelined execution mode with caution.

3.11. Non-blocking TLS/SSL

3.11.1. SSL I/O session

`SSLIOSession` is a decorator class intended to transparently extend any arbitrary `IOSession` with transport layer security capabilities based on the SSL/TLS protocol. Default HTTP connection implementations and protocol handlers should be able to work with SSL sessions without special preconditions or modifications.

```
SSLContext sslcontext = SSLContext.getInstance("Default");
sslcontext.init(null, null, null);
// Plain I/O session
IOSession iosession = <...>
SSLIOSession sslsession = new SSLIOSession(
        iosession, SSLMode.CLIENT, sslcontext, null);
iosession.setAttribute(SSLIOSession.SESSION_KEY, sslsession);
NHttpClientConnection conn = new DefaultNHttpClientConnection(
        sslsession, 8 * 1024);
```

One can also use `SSLNHttpClientConnectionFactory` or `SSLNHttpServerConnectionFactory` classes to conveniently create SSL encrypterd HTTP connections.

```
SSLContext sslcontext = SSLContext.getInstance("Default");
sslcontext.init(null, null, null);
// Plain I/O session
IOSession iosession = <...>
SSLNHttpClientConnectionFactory connfactory = new SSLNHttpClientConnectionFactory(
        sslcontext, null, ConnectionConfig.DEFAULT);
NHttpClientConnection conn = connfactory.createConnection(iosession);
```

3.11.1.1. SSL setup handler

Applications can customize various aspects of the TLS/SSl protocol by passing a custom implementation of the `SSLSetupHandler` interface.

SSL events as defined by the `SSLSetupHandler` interface:

- **`initalize`:** Triggered when the SSL connection is being initialized. The handler can use this callback to customize properties of the `javax.net.ssl.SSLEngine` used to establish the SSL session.
- **`verify`:** Triggered when the SSL connection has been established and initial SSL handshake has been successfully completed. The handler can use this callback to verify properties of the SSLSession. For instance this would be the right place to enforce SSL cipher strength, validate certificate chain and do hostname checks.

```
```

```
SSLContext sslcontext = SSLContexts.createDefault();
// Plain I/O session
IOSession iosession = <...>

SSLIOSession sslsession = new SSLIOSession(
        iosession, SSLMode.CLIENT, sslcontext, new SSLSetupHandler() {

    public void initalize(final SSLEngine sslengine) throws SSLException {
        // Enforce TLS and disable SSL
        sslengine.setEnabledProtocols(new String[] {
                "TLSv1",
                "TLSv1.1",
                "TLSv1.2" });
        // Enforce strong ciphers
        sslengine.setEnabledCipherSuites(new String[] {
                "TLS_RSA_WITH_AES_256_CBC_SHA",
                "TLS_DHE_RSA_WITH_AES_256_CBC_SHA",
                "TLS_DHE_DSS_WITH_AES_256_CBC_SHA" });
    }

    public void verify(
            final IOSession iosession,
            final SSLSession sslsession) throws SSLException {
        X509Certificate[] certs = sslsession.getPeerCertificateChain();
        // Examine peer certificate chain
        for (X509Certificate cert: certs) {
            System.out.println(cert.toString());
        }
    }

});
```

`SSLSetupHandler` impelemntations can also be used with the `SSLNHttpClientConnectionFactory` or `SSLNHttpServerConnectionFactory` classes.

```

SSLContext sslcontext = SSLContexts.createDefault();
// Plain I/O session
IOSession iosession = <...>

SSLSetupHandler mysslhandler = new SSLSetupHandler() {

    public void initalize(final SSLEngine sslengine) throws SSLException {
        // Enforce TLS and disable SSL
        sslengine.setEnabledProtocols(new String[] {
                "TLSv1",
                "TLSv1.1",
                "TLSv1.2" });
    }

    public void verify(
            final IOSession iosession, final SSLSession sslsession) throws SSLException {
    }


};
SSLNHttpClientConnectionFactory connfactory = new SSLNHttpClientConnectionFactory(
        sslcontext, mysslhandler, ConnectionConfig.DEFAULT);
NHttpClientConnection conn = connfactory.createConnection(iosession);
```

3.11.2. TLS/SSL aware I/O event dispatches

Default `IOEventDispatch` implementations shipped with the library such as `DefaultHttpServerIODispatch` and `DefaultHttpClientIODispatch` automatically detect SSL

encrypted sessions and handle SSL transport aspects transparently. However, custom I/O event dispatchers that do not extend `AbstractIODispatch` are required to take some additional actions to ensure correct functioning of the transport layer encryption.

- The I/O dispatch may need to call `SSLIOSession#initalize()` method in order to put the SSL session either into a client or a server mode, if the SSL session has not been yet initialized.
- When the underlying I/O session is input ready, the I/O dispatcher should check whether the SSL I/O session is ready to produce input data by calling `SSLIOSession#isAppInputReady()`, pass control to the protocol handler if it is, and finally call `SSLIOSession#inboundTransport()` method in order to do the necessary SSL handshaking and decrypt input data.
- When the underlying I/O session is output ready, the I/O dispatcher should check whether the SSL I/O session is ready to accept output data by calling `SSLIOSession#isAppOutputReady()`, pass control to the protocol handler if it is, and finally call `SSLIOSession#outboundTransport()` method in order to do the necessary SSL handshaking and encrypt application data.

3.12. Embedded non-blocking HTTP server

As of version 4.4 HttpCore ships with an embedded non-blocking HTTP server based on non-blocking I/O components described above.

```
HttpAsyncRequestHandler<?> requestHandler = <...>
HttpProcessor httpProcessor = <...>
SocketConfig socketConfig = SocketConfig.custom()
        .setSoTimeout(15000)
        .setTcpNoDelay(true)
        .build();
final HttpServer server = ServerBootstrap.bootstrap()
        .setListenerPort(8080)
        .setHttpProcessor(httpProcessor)
        .setSocketConfig(socketConfig)
        .setExceptionLogger(new StdErrorExceptionLogger())
        .registerHandler("*", requestHandler)
        .create();
server.start();
server.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);

Runtime.getRuntime().addShutdownHook(new Thread() {
    @Override
    public void run() {
        server.shutdown(5, TimeUnit.SECONDS);
    }
});
```

Chapter 4. Advanced topics

4.1. HTTP message parsing and formatting framework

HTTP message processing framework is designed to be expressive and flexible while remaining memory efficient and fast. HttpCore HTTP message processing code achieves near zero intermediate garbage and near zero-copy buffering for its parsing and formatting operations. The same HTTP message parsing and formatting API and implementations are used by both the blocking and non-blocking transport implementations, which helps ensure a consistent behavior of HTTP services regardless of the I/O model.

4.1.1. HTTP line parsing and formatting

HttpCore utilizes a number of low level components for all its line parsing and formatting methods.

`CharArrayBuffer` represents a sequence of characters, usually a single line in an HTTP message stream such as a request line, a status line or a header. Internally `CharArrayBuffer` is backed by an array of chars, which can be expanded to accommodate more input if needed. `CharArrayBuffer` also provides a number of utility methods for manipulating content of the buffer, storing more data and retrieving subsets of data.

```
CharArrayBuffer buf = new CharArrayBuffer(64);
buf.append("header:  data ");
int i = buf.indexOf(':');
String s = buf.substringTrimmed(i + 1, buf.length());
System.out.println(s);
System.out.println(s.length());
```

stdout >

```
data
4
```

`ParserCursor` represents a context of a parsing operation: the bounds limiting the scope of the parsing operation and the current position the parsing operation is expected to start at.

```
CharArrayBuffer buf = new CharArrayBuffer(64);
buf.append("header:  data ");
int i = buf.indexOf(':');
ParserCursor cursor = new ParserCursor(0, buf.length());
cursor.updatePos(i + 1);
System.out.println(cursor);
```

stdout >

```
[0>7>14]
```

`LineParser` is the interface for parsing lines in the head section of an HTTP message. There are individual methods for parsing a request line, a status line, or a header line. The lines to parse are passed in-memory, the parser does not depend on any specific I/O mechanism.

```
CharArrayBuffer buf = new CharArrayBuffer(64);
buf.append("HTTP/1.1 200");
ParserCursor cursor = new ParserCursor(0, buf.length());

LineParser parser = BasicLineParser.INSTANCE;
ProtocolVersion ver = parser.parseProtocolVersion(buf, cursor);
System.out.println(ver);
System.out.println(buf.substringTrimmed(
        cursor.getPos(),
        cursor.getUpperBound()));
```

stdout >

```
HTTP/1.1
200
```

```
CharArrayBuffer buf = new CharArrayBuffer(64);
buf.append("HTTP/1.1 200 OK");
ParserCursor cursor = new ParserCursor(0, buf.length());
LineParser parser = new BasicLineParser();
StatusLine sl = parser.parseStatusLine(buf, cursor);
System.out.println(sl.getReasonPhrase());
```

stdout >

```
OK
```

`LineFormatter` for formatting elements of the head section of an HTTP message. This is the complement to `LineParser` . There are individual methods for formatting a request line, a status line, or a header line.

Please note the formatting does not include the trailing line break sequence `CR-LF`.

```
CharArrayBuffer buf = new CharArrayBuffer(64);
LineFormatter formatter = new BasicLineFormatter();
formatter.formatRequestLine(buf,
    new BasicRequestLine("GET", "/", HttpVersion.HTTP_1_1));
System.out.println(buf.toString());
formatter.formatHeader(buf,
    new BasicHeader("Content-Type", "text/plain"));
System.out.println(buf.toString());
```

stdout >

```
GET / HTTP/1.1
Content-Type: text/plain
```

`HeaderValueParser` is the interface for parsing header values into elements.

```
CharArrayBuffer buf = new CharArrayBuffer(64);
HeaderValueParser parser = new BasicHeaderValueParser();
buf.append("name1=value1; param1=p1, " +
    "name2 = \"value2\", name3  = value3");
ParserCursor cursor = new ParserCursor(0, buf.length());
System.out.println(parser.parseHeaderElement(buf, cursor));
```

```
System.out.println(parser.parseHeaderElement(buf, cursor));
System.out.println(parser.parseHeaderElement(buf, cursor));
```

stdout >

```
name1=value1; param1=p1
name2=value2
name3=value3
```

`HeaderValueFormatter` is the interface for formatting elements of a header value. This is the complement to `HeaderValueParser` .

```
CharArrayBuffer buf = new CharArrayBuffer(64);
HeaderValueFormatter formatter = new BasicHeaderValueFormatter();
HeaderElement[] hes = new HeaderElement[] {
        new BasicHeaderElement("name1", "value1",
                new NameValuePair[] {
                    new BasicNameValuePair("param1", "p1")} ),
        new BasicHeaderElement("name2", "value2"),
        new BasicHeaderElement("name3", "value3"),
};
formatter.formatElements(buf, hes, true);
System.out.println(buf.toString());
```

stdout >

```
name1="value1"; param1="p1", name2="value2", name3="value3"
```

4.1.2. HTTP message streams and session I/O buffers

HttpCore provides a number of utility classes for the blocking and non-blocking I/O models that facilitate the processing of HTTP message streams, simplify handling of `CR-LF` delimited lines in HTTP messages and manage intermediate data buffering.

HTTP connection implementations usually rely on session input/output buffers for reading and writing data from and to an HTTP message stream. Session input/output buffer implementations are I/O model specific and are optimized either for blocking or non-blocking operations.

Blocking HTTP connections use socket bound session buffers to transfer data. Session buffer interfaces are similar to `java.io.InputStream` / `java.io.OutputStream` classes, but they also provide methods for reading and writing `CR-LF` delimited lines.

```
Socket socket1 = <...>
Socket socket2 = <...>
HttpTransportMetricsImpl metrics = new HttpTransportMetricsImpl();
SessionInputBufferImpl inbuffer = new SessionInputBufferImpl(metrics, 8 * 1024);
inbuffer.bind(socket1.getInputStream());
SessionOutputBufferImpl outbuffer = new SessionOutputBufferImpl(metrics, 8 * 1024);
outbuffer.bind(socket2.getOutputStream());
CharArrayBuffer linebuf = new CharArrayBuffer(1024);
inbuffer.readLine(linebuf);
outbuffer.writeLine(linebuf);
```

Non-blocking HTTP connections use session buffers optimized for reading and writing data from and to non-blocking NIO channels. NIO session input/output sessions help deal with `CR-LF` delimited lines in a non-blocking I/O mode.

```
ReadableByteChannel channel1 = <...>
WritableByteChannel channel2 = <...>

SessionInputBuffer inbuffer = new SessionInputBufferImpl(8 * 1024);
SessionOutputBuffer outbuffer = new SessionOutputBufferImpl(8 * 1024);

CharArrayBuffer linebuf = new CharArrayBuffer(1024);
boolean endOfStream = false;
int bytesRead = inbuffer.fill(channel1);
if (bytesRead == -1) {
    endOfStream = true;
}
if (inbuffer.readLine(linebuf, endOfStream)) {
    outbuffer.writeLine(linebuf);
}
if (outbuffer.hasData()) {
    outbuffer.flush(channel2);
}
```

4.1.3. HTTP message parsers and formatters

HttpCore also provides coarse-grained facade type interfaces for parsing and formatting of HTTP messages. Default implementations of those interfaces build upon the functionality provided by `SessionInputBuffer` / `SessionOutputBuffer` and `HttpLineParser` / `HttpLineFormatter` implementations.

Example of HTTP request parsing / writing for blocking HTTP connections:

```
SessionInputBuffer inbuffer = <...>
SessionOutputBuffer outbuffer = <...>

HttpMessageParser<HttpRequest> requestParser = new DefaultHttpRequestParser(
    inbuffer);
HttpRequest request = requestParser.parse();
HttpMessageWriter<HttpRequest> requestWriter = new DefaultHttpRequestWriter(
    outbuffer);
requestWriter.write(request);
```

Example of HTTP response parsing / writing for blocking HTTP connections:

```
SessionInputBuffer inbuffer = <...>
SessionOutputBuffer outbuffer = <...>

HttpMessageParser<HttpResponse> responseParser = new DefaultHttpResponseParser(
        inbuffer);
HttpResponse response = responseParser.parse();
HttpMessageWriter<HttpResponse> responseWriter = new DefaultHttpResponseWriter(
        outbuffer);
responseWriter.write(response);
```

Custom message parsers and writers can be plugged into the message processing pipeline through a custom connection factory:

```
HttpMessageWriterFactory<HttpResponse> responseWriterFactory =
                                new HttpMessageWriterFactory<HttpResponse>() {
    @Override
    public HttpMessageWriter<HttpResponse> create(
            SessionOutputBuffer buffer) {
```

```
        HttpMessageWriter<HttpResponse> customWriter = <...>
        return customWriter;
    }
};
HttpMessageParserFactory<HttpRequest> requestParserFactory =
                                new HttpMessageParserFactory<HttpRequest>() {
    @Override
    public HttpMessageParser<HttpRequest> create(
            SessionInputBuffer buffer,
            MessageConstraints constraints) {
        HttpMessageParser<HttpRequest> customParser = <...>
        return customParser;
    }
};
HttpConnectionFactory<DefaultBHttpServerConnection> cf =
                                new DefaultBHttpServerConnectionFactory(
        ConnectionConfig.DEFAULT,
        requestParserFactory,
        responseWriterFactory);
Socket socket = <...>
DefaultBHttpServerConnection conn = cf.createConnection(socket);
```

Example of HTTP request parsing / writing for non-blocking HTTP connections:

```

SessionInputBuffer inbuffer = <...>
SessionOutputBuffer outbuffer  = <...>

NHttpMessageParser<HttpRequest> requestParser = new DefaultHttpRequestParser(
        inbuffer);
HttpRequest request = requestParser.parse();
NHttpMessageWriter<HttpRequest> requestWriter = new DefaultHttpRequestWriter(
        outbuffer);
requestWriter.write(request);
```

Example of HTTP response parsing / writing for non-blocking HTTP connections:

```

SessionInputBuffer inbuffer = <...>
SessionOutputBuffer outbuffer  = <...>

NHttpMessageParser<HttpResponse> responseParser = new DefaultHttpResponseParser(
        inbuffer);
HttpResponse response = responseParser.parse();
NHttpMessageWriter responseWriter = new DefaultHttpResponseWriter(
        outbuffer);
responseWriter.write(response);
```

Custom non-blocking message parsers and writers can be plugged into the message processing pipeline through a custom connection factory:

```

NHttpMessageWriterFactory<HttpResponse> responseWriterFactory =
                        new NHttpMessageWriterFactory<HttpResponse>() {
    @Override
    public NHttpMessageWriter<HttpResponse> create(SessionOutputBuffer buffer) {
        NHttpMessageWriter<HttpResponse> customWriter = <...>
        return customWriter;
    }
};
NHttpMessageParserFactory<HttpRequest> requestParserFactory =
                        new NHttpMessageParserFactory<HttpRequest>() {
    @Override
    public NHttpMessageParser<HttpRequest> create(
            SessionInputBuffer buffer, MessageConstraints constraints) {
```

```
        NHttpMessageParser<HttpRequest> customParser = <...>
        return customParser;
    }
};
NHttpConnectionFactory<DefaultNHttpServerConnection> cf =
                        new DefaultNHttpServerConnectionFactory(
        null,
        requestParserFactory,
        responseWriterFactory,
        ConnectionConfig.DEFAULT);
IOSession iosession = <...>
DefaultNHttpServerConnection conn = cf.createConnection(iosession);
```

4.1.4. HTTP header parsing on demand

The default implementations of `HttpMessageParser` and `NHttpMessageParser` interfaces do not parse HTTP headers immediately. Parsing of header value is deferred until its properties are accessed. Those headers that are never used by the application will not be parsed at all. The `CharArrayBuffer` backing the header can be obtained through an optional `FormattedHeader` interface.

```
HttpResponse response = <...>
Header h1 = response.getFirstHeader("Content-Type");
if (h1 instanceof FormattedHeader) {
    CharArrayBuffer buf = ((FormattedHeader) h1).getBuffer();
    System.out.println(buf);
}
```