HttpCore Tutorial

Oleg Kalnichevski

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 a Servlet container or a competitor to the Servlet API

Chapter 1. Fundamentals

1.1. HTTP messages

1.1.1. Structure

A HTTP message consists of a head and an optional body. The message head of an HTTP request consists of a request line and a collection of header fields. The message head 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 that closely follows the definition and provides an 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 be applied 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 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 get tokenized into individual header elements only on demand. HTTP headers received over an HTTP connection are stored internally as an array of chars and parsed lazily only when their properties are accessed.

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.

This distinction is important for connection management with incoming entities. For entities that are created by an application and only sent using the HttpCore framework, the difference between streamed and self-contained is of little importance. In that case, it is suggested to consider non-repeatable entities as streamed, and those that are repeatable as self-contained.

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, ie. 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.

The `EntityUtils` class exposes several static methods to more easily read the content or information from an entity. Instead of reading the `java.io.InputStream` directly, one can retrieve the whole content body in a string / 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",
    "UTF-8");

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

stdout >

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

1.1.3.3. Ensuring release of low level resources

When finished with an entity that relies on an underlying input stream, it's important to execute the `HttpEntity#consumeContent()` method, so as to consume any available content on the stream, so the connection could be released to any connection pools. If the incoming content is not consumed fully, other requests may fail when this connection is re-used.

Alternatively one can simply check the result of `HttpEntity#isStreaming()`, and keep reading from the input stream until it returns false.

Self contained entities will always return false with `HttpEntity#isStreaming()`, as there is no underlying stream it depends on. For these entities `HttpEntity#consumeContent()` will do nothing, and does not need to be called.

1.1.4. Creating entities

There are a few ways to create entities. The following implementations are provided by HttpCore:

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

1.1.4.1. `BasicHttpEntity`

This is exactly as the name implies, a basic entity that represents an underlying stream. This is generally used for the 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. This byte array is supplied to the constructor.

```
String myData = "Hello world on the other side!!";
ByteArrayEntity myEntity = new ByteArrayEntity(myData.getBytes());
```

1.1.4.3. `StringEntity`

`StringEntity` is a self contained, repeatable entity that obtains its content from a `java.lang.String` object. It has two constructors, one simply constructs with a given `java.lang.String` object; the other also takes a character encoding for the data in the string.

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

// construct without a character encoding
HttpEntity myEntity1 = new StringEntity(sb.toString());

// alternatively construct with an encoding
HttpEntity myEntity2 = new StringEntity(sb.toString(), "UTF-8");
```

1.1.4.4. `InputStreamEntity`

`InputStreamEntity` is a streamed, non-repeatable entity that obtains its content from an input stream. It is constructed by supplying the input stream and the content length. The content length is used 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 the length is most often used to limit the length.

```
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. Since this is mostly used to stream large files of different types, one needs 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,
    "application/java-archive");
```

1.1.4.6. `EntityTemplate`

This is an entity which receives its content from a `ContentProducer` interface. Content producers are objects which produce their content on demand, by writing it out to an output stream. They are expected to be able produce their content every time they are requested to do so. So creating a `EntityTemplate`, one is expected to supply a reference to a content producer, which effectively creates a repeatable entity.

There are no standard content producers in HttpCore. It is basically just a convenience interface to allow wrapping up complex logic into an entity. To use this entity one needs to create a class that implements `ContentProducer` and override the `ContentProducer#writeTo(OutputStream)` method. Then, an instance of custom `ContentProducer` will be used to write the full content body to the output stream. For instance, an HTTP server would serve static files with the `FileEntity`, but running CGI programs could be done with a `ContentProducer`, inside which one could implement custom logic to supply the content as it becomes available. This way one does not need to buffer it in a string and then use a `StringEntity` or `ByteArrayEntity`.

```
ContentProducer myContentProducer = new ContentProducer() {
```

```
    public void writeTo(OutputStream out) throws IOException {
      out.write("ContentProducer rocks! ".getBytes());
      out.write(("Time requested: " + new Date()).getBytes());
    }

};

HttpEntity myEntity = new EntityTemplate(myContentProducer);
myEntity.writeTo(System.out);
```

stdout >

```
ContentProducer rocks! Time requested: Fri Sep 05 12:20:22 CEST 2008
```

1.1.4.7. `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.

1.1.4.8. `BufferedHttpEntity`

`BufferedHttpEntity` is a subclass of `HttpEntityWrapper`. It is constructed 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, calls are simply passed through to the underlying entity.

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

1.2. 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. It is strongly recommended to limit 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()` .

1.2.1. 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 = new Socket();
// Initialize socket
BasicHttpParams params = new BasicHttpParams();
DefaultHttpClientConnection conn = new DefaultHttpClientConnection();
conn.bind(socket, params);
conn.isOpen();
HttpConnectionMetrics metrics = conn.getMetrics();
metrics.getRequestCount();
metrics.getResponseCount();
metrics.getReceivedBytesCount();
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 it may be followed by a message body. Please note it is very important to call `HttpEntity#consumeContent()` 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 the content of the message body is fully consumed for that connection to be potentially re-usable.

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

```
Socket socket = new Socket();
// Initialize socket
HttpParams params = new BasicHttpParams();
DefaultHttpClientConnection conn = new DefaultHttpClientConnection();
conn.bind(socket, params);
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, call
    // consumeContent() to make sure the connection can be re-used
    entity.consumeContent();
}
```

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

```
Socket socket = new Socket();
// Initialize socket
HttpParams params = new BasicHttpParams();
DefaultHttpServerConnection conn = new DefaultHttpServerConnection();
conn.bind(socket, params);
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,
        // call consumeContent() to make sure the connection
        // can be re-used
        entity.consumeContent();
    }
}
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 use appropriate higher level HTTP service implementations instead.

1.2.2. 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 completely transparently for the data consumer. Likewise, HTTP connections rely on `HttpEntity#writeTo(OutputStream)` method to generate the content of an outgoing message. If an outgoing messages encloses an entity, the content will be encoded automatically based on the properties of the message.

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

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

1.2.4. 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.

1.3. 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.

1.3.1. Protocol exception

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

1.4. 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.

1.4.1. Standard protocol interceptors

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

1.4.1.1. `RequestContent`

`RequestContent` is the most important interceptor for outgoing requests. 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 client side protocol processors.

1.4.1.2. `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.

1.4.1.3. `RequestConnControl`

`RequestConnControl` is responsible for adding `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.

1.4.1.4. `ResponseConnControl`

`ResponseConnControl` is responsible for adding `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.

1.4.1.5. `RequestDate`

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

1.4.1.6. `ResponseDate`

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

1.4.1.7. `RequestExpectContinue`

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

1.4.1.8. `RequestTargetHost`

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

1.4.1.9. `RequestUserAgent`

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

1.4.1.10. `ResponseServer`

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

1.4.2. 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.

```
BasicHttpProcessor httpproc = new BasicHttpProcessor();
// Required protocol interceptors
httpproc.addInterceptor(new RequestContent());
httpproc.addInterceptor(new RequestTargetHost());
// Recommended protocol interceptors
httpproc.addInterceptor(new RequestConnControl());
httpproc.addInterceptor(new RequestUserAgent());
httpproc.addInterceptor(new RequestExpectContinue());

HttpContext context = new BasicHttpContext();

HttpRequest request = new BasicHttpRequest("GET", "/");
httpproc.process(request, context);
HttpResponse response = null;
```

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

```
httpproc.process(response, context);
```

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

1.4.3. HTTP context

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.

```
BasicHttpProcessor httpproc = new BasicHttpProcessor();
httpproc.addInterceptor(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);
        }
    }


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

`HttpContext` instances can be linked together to form a hierarchy. In the simplest form one context can use content of another context to obtain default values of attributes not present in the local context.

```
HttpContext parentContext = new BasicHttpContext();
parentContext.setAttribute("param1", Integer.valueOf(1));
parentContext.setAttribute("param2", Integer.valueOf(2));

HttpContext localContext = new BasicHttpContext();
localContext.setAttribute("param2", Integer.valueOf(0));
localContext.setAttribute("param3", Integer.valueOf(3));
HttpContext stack = new DefaultedHttpContext(localContext,
    parentContext);

System.out.println(stack.getAttribute("param1"));
System.out.println(stack.getAttribute("param2"));
System.out.println(stack.getAttribute("param3"));
System.out.println(stack.getAttribute("param4"));
```

stdout >

```
1
0
```

```
3
null
```

1.5. HTTP parameters

`HttpParams` interface represents a collection of immutable values that define a runtime behavior of a component. In many ways `HttpParams` is similar to `HttpContext`. The main distinction between the two lies in their use at runtime. Both interfaces represent a collection of objects that are organized as a map of textual names to object values, but serve distinct purposes:

- `HttpParams` is intended to contain simple objects: integers, doubles, strings, collections and objects that remain immutable at runtime. `HttpParams` is expected to be used in the 'write once - ready many' mode. `HttpContext` is intended to contain complex objects that are very likely to mutate in the course of HTTP message processing.

- The purpose of `HttpParams` is to define a behavior of other components. Usually each complex component has its own `HttpParams` object. The purpose of `HttpContext` is to represent an execution state of an HTTP process. Usually the same execution context is shared among many collaborating objects.

`HttpParams`, like `HttpContext` can be linked together to form a hierarchy. In the simplest form one set of parameters can use content of another one to obtain default values of parameters not present in the local set.

```
HttpParams parentParams = new BasicHttpParams();
parentParams.setParameter(CoreProtocolPNames.PROTOCOL_VERSION,
    HttpVersion.HTTP_1_0);
parentParams.setParameter(CoreProtocolPNames.HTTP_CONTENT_CHARSET,
    "UTF-8");

HttpParams localParams = new BasicHttpParams();
localParams.setParameter(CoreProtocolPNames.PROTOCOL_VERSION,
    HttpVersion.HTTP_1_1);
localParams.setParameter(CoreProtocolPNames.USE_EXPECT_CONTINUE,
    Boolean.FALSE);
HttpParams stack = new DefaultedHttpParams(localParams,
    parentParams);

System.out.println(stack.getParameter(
    CoreProtocolPNames.PROTOCOL_VERSION));
System.out.println(stack.getParameter(
    CoreProtocolPNames.HTTP_CONTENT_CHARSET));
System.out.println(stack.getParameter(
    CoreProtocolPNames.USE_EXPECT_CONTINUE));
System.out.println(stack.getParameter(
    CoreProtocolPNames.USER_AGENT));
```

stdout >

```
HTTP/1.1
UTF-8
false
null
```

Please note the `BasicHttpParams` class does not synchronize access to its internal structures and therefore amy be thread-unsafe.

1.5.1. HTTP parameter beans

`HttpParams` interface allows for a great deal of flexibility in handling configuration of components. Most importantly, new parameters can be introduced without affecting binary compatibility with older versions. However, `HttpParams` also has a certain disadvantage compared to regular Java beans: `HttpParams` cannot be assembled using a DI framework. To mitigate the limitation, HttpCore includes a number of bean classes that can used in order to initialize `HttpParams` objects using standard Java bean conventions.

```
HttpParams params = new BasicHttpParams();
HttpProtocolParamBean paramsBean = new HttpProtocolParamBean(params);
paramsBean.setVersion(HttpVersion.HTTP_1_1);
paramsBean.setContentCharset("UTF-8");
paramsBean.setUseExpectContinue(true);

System.out.println(params.getParameter(
    CoreProtocolPNames.PROTOCOL_VERSION));
System.out.println(params.getParameter(
    CoreProtocolPNames.HTTP_CONTENT_CHARSET));
System.out.println(params.getParameter(
    CoreProtocolPNames.USE_EXPECT_CONTINUE));
System.out.println(params.getParameter(
    CoreProtocolPNames.USER_AGENT));
```

stdout >

```
HTTP/1.1
UTF-8
false
null
```

1.6. Blocking HTTP protocol handlers

1.6.1. 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.

```
HttpParams params;
// Initialize HTTP parameters
HttpProcessor httpproc;
// Initialize HTTP processor

HttpService httpService = new HttpService(
        httpproc,
        new DefaultConnectionReuseStrategy(),
        new DefaultHttpResponseFactory());
httpService.setParams(params);
```

1.6.1.1. 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.addHeader("Content-Type", "text/plain");
        response.setEntity(
            new StringEntity("some important message"));
    }

};
```

1.6.1.2. 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>`.

```
HttpService httpService;
// Initialize HTTP service

HttpRequestHandlerRegistry handlerResolver =
    new HttpRequestHandlerRegistry();
handlerReqistry.register("/service/*", myRequestHandler1);
handlerReqistry.register("*.do", myRequestHandler2);
handlerReqistry.register("*", myRequestHandler3);

// Inject handler resolver
httpService.setHandlerResolver(handlerResolver);
```

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

1.6.1.3. 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;
// Initialize HTTP service
```

```
HttpServerConnection conn;
// Initialize connection
HttpContext context;
// Initialize HTTP context

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

1.6.2. 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. `HttpRequestExecutor` relies on 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. Application specific processing can be implemented outside `HttpRequestExecutor` once the request has been executed and a response has been received.

```
HttpClientConnection conn;
// Create connection
HttpParams params;
// Initialize HTTP parameters
HttpProcessor httpproc;
// Initialize HTTP processor
HttpContext context;
// Initialize HTTP context

HttpRequestExecutor httpexecutor = new HttpRequestExecutor();

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

HttpEntity entity = response.getEntity();
if (entity != null) {
    entity.consumeContent();
}
```

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.

1.6.3. 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.

Chapter 2. NIO extensions

2.1. Benefits and shortcomings of the non-blocking I/O model

Contrary to the popular belief, the performance of NIO in terms of raw data throughput is significantly lower than that of blocking I/O. NIO does not necessarily fit all use cases and should be used only where appropriate:

- handling of thousands of connections, a significant number of which can be idle.
- handling high latency connections.
- request / response handling needs to be decoupled.

2.2. Differences from other NIO 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 only as much input data as it can process without having to allocate more memory.
- direct access to the NIO channels where possible.

2.3. 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 implementing 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.

```
HttpParams params = new BasicHttpParams();
int workerCount = 2;
IOReactor ioreactor = new DefaultConnectingIOReactor(workerCount,
    params);
```

2.3.1. 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.

```
HttpParams params = new BasicHttpParams();
IOReactor ioreactor = new DefaultConnectingIOReactor(2, params);

IOEventDispatch eventDispatch = new MyIOEventDispatch();
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.

2.3.2. 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.

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

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

2.3.3. 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);
```

2.3.4. 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);
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.

2.3.5. 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.setEvent(SelectionKey.OP_READ);
iosession.clearEvent(SelectionKey.OP_READ);
```

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

2.3.6. 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 consumer can read 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 = new MySessionBufferStatus();
iosession.setBufferStatus(myBufferStatus);
iosession.hasBufferedInput();
iosession.hasBufferedOutput();
```

2.3.7. 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.

2.3.8. 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();
    }
}
```

2.3.9. 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) {
            }

        });
```

2.4. 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 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 behaviour 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.

2.4.1. 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();
}
```

2.5. 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.

2.5.1. 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.

```
// Get non-blocking HTTP connection
DefaultNHttpClientConnection conn;
// State
Object myStateObject;

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

2.5.2. 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:

```
// Obtain HTTP connection
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:

```
// Obtain HTTP connection
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.

2.5.3. 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.

2.5.4. 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.

```
//Obtain content decoder
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.

```
// Obtain content encoder
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.

```
// Obtain HTTP connection
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.

```
// Obtain HTTP connection
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());
}
```

2.5.5. 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.

2.5.6. 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 such as `Content-Length` delimited and identity 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`.

```
//Obtain content decoder
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`.

```
// Obtain content encoder
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();
}
```

2.6. 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 `NHttpClientHandler` and `NHttpServiceHandler` interfaces to propagate HTTP protocol events to a HTTP protocol handler.

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

- **`connected`:** Triggered when a 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 be temporarily suspended using `IOControl` interface.
- **`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.
- **`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.
- **`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 `NHttpClientHandler` 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.
- **`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.
- **`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..
- **`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.

2.7. Non-blocking HTTP entities

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 applicable to the asynchronous process of data transfer. Therefore, non-blocking HTTP entities provide NIO specific extensions to the !HttpEntity interface: `ProducingNHttpEntity` and `ConsumingNHttpEntity` interfaces. Implementation classes of these interfaces may throw `java.lang.UnsupportedOperationException` from `HttpEntity#getContent()` or `HttpEntity#writeTo()` if a particular implementation is unable to represent its content stream as instance of `java.io.InputStream` or cannot stream its content out to an `java.io.OutputStream`.

2.7.1. Content consuming non-blocking HTTP entity

`ConsumingNHttpEntity` interface represents a non-blocking entity that allows content to be consumed from a content decoder. `ConsumingNHttpEntity` extends the base `HttpEntity` interface with a number of NIO specific notification methods:

- **`consumeContent`:** Notification that content is available to be read from the decoder. `IOControl` instance passed as a parameter to the method can be used to suspend input events if the entity is temporarily unable to allocate more storage to accommodate all incoming content.
- **`finish`:** Notification that any resources allocated for reading can be released.

The following implementations of `ConsumingNHttpEntity` provided by HttpCore NIO:

- `BufferingNHttpEntity`
- `ConsumingNHttpEntityTemplate`

2.7.1.1. `BufferingNHttpEntity`

`BufferingNHttpEntity` is a subclass of `HttpEntityWrapper` that consumes all incoming content into memory. Once the content body has been fully received it can be retrieved as an `java.io.InputStream` via `HttpEntity#getContent()` , or written to an output stream via `HttpEntity#writeTo()` .

2.7.1.2. `ConsumingNHttpEntityTemplate`

`ConsumingNHttpEntityTemplate` is a subclass of `HttpEntityWrapper` that that decorates the incoming HTTP entity and delegates the handling of incoming content to a `ContentListener` instance.

```
static class FileWriteListener implements ContentListener {

    private final FileChannel fileChannel;
    private long idx = 0;

    public FileWriteListener(File file) throws IOException {
        this.fileChannel = new FileInputStream(file).getChannel();
    }

    public void contentAvailable(
    ContentDecoder decoder, IOControl ioctrl) throws IOException {
        long transferred;
        if (decoder instanceof FileContentDecoder) {
            transferred = ((FileContentDecoder) decoder).transfer(
                    fileChannel, idx, Long.MAX_VALUE);
        } else {
            transferred = fileChannel.transferFrom(
                    new ContentDecoderChannel(decoder),
                    idx, Long.MAX_VALUE);
        }
        if (transferred > 0) {
            idx += transferred;
        }
    }

    public void finished() {
        try {
            fileChannel.close();
        } catch(IOException ignored) {}
    }

}

HttpEntity incomingEntity;

File file = new File("buffer.bin");
ConsumingNHttpEntity entity = new ConsumingNHttpEntityTemplate(
```

```
        incomingEntity,
        new FileWriteListener(file));
```

2.7.2. Content producing non-blocking HTTP entity

`ProducingNHttpEntity` interface represents a non-blocking entity that allows content to be written to a content encoder. `ProducingNHttpEntity` extends the base `HttpEntity` interface with a number of NIO specific notification methods:

- **`produceContent:`** Notification that content can be written to the encoder. `IOControl` instance passed as a parameter to the method can be used to temporarily suspend output events if the entity is unable to produce more content. Please note one must call `ContentEncoder#complete()` to inform the underlying connection that all content has been written. Failure to do so could result in the entity never being correctly delimited.
- **`finish:`** Notification that any resources allocated for writing can be released.

The following implementations of `ProducingNHttpEntity` provided by HttpCore NIO:

- `NByteArrayEntity`
- `NStringEntity`
- `NFileEntity`

2.7.2.1. `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.

```
String myData = "Hello world on the other side!!";
NByteArrayEntity entity = new NByteArrayEntity(myData.getBytes());
```

2.7.2.2. `NStringEntity`

It's 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`.

```
String myData = "Hello world on the other side!!";
// construct without a character encoding
NStringEntity myEntity1 = new NStringEntity(myData);
// alternatively construct with an encoding
NStringEntity myEntity2 = new NStringEntity(myData, "UTF-8");
```

2.7.2.3. `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");
NHttpEntity entity = new NFileEntity(staticFile,
```

```
    "application/java-archive");
```

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.

2.8. Non-blocking HTTP protocol handlers

2.8.1. Asynchronous HTTP service handler

`AsyncNHttpServiceHandler` is a fully asynchronous HTTP server side protocol handler that implements the essential requirements of the HTTP protocol for the server side message processing as described by RFC 2616. `AsyncNHttpServiceHandler` is capable of processing HTTP requests with nearly constant memory footprint for individual HTTP connections. The handler stores headers of HTTP messages in memory, while content of message bodies is streamed directly from the entity to the underlying channel (and vice versa) using `ConsumingNHttpEntity` and `ProducingNHttpEntity` interfaces.

When using this implementation, it is important to ensure that entities supplied for writing implement `ProducingNHttpEntity`. Doing so will allow the entity to be written out asynchronously. If entities supplied for writing do not implement the `ProducingNHttpEntity` interface, a delegate is added that buffers the entire contents in memory. Additionally, the buffering might take place in the I/O dispatch thread, which could cause I/O to block temporarily. For best results, one must ensure that all entities set on HTTP responses from `NHttpRequestHandler` implement `ProducingNHttpEntity`.

If incoming requests enclose a content entity, `NHttpRequestHandler` instances are expected to return a `ConsumingNHttpEntity` for reading the content. After the entity is finished reading the data, `NHttpRequestHandler#handle()` method is called to generate a response.

`AsyncNHttpServiceHandler` 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 take care of application specific content generation and processing.

```
HttpParams params;
// Initialize HTTP parameters
HttpProcessor httpproc;
// Initialize HTTP processor

AsyncNHttpServiceHandler handler = new AsyncNHttpServiceHandler(
        httpproc,
        new DefaultHttpResponseFactory(),
        new DefaultConnectionReuseStrategy(),
        params);
```

2.8.1.1. Non-blocking HTTP request handlers

`NHttpRequestHandler` interface represents a routine for processing of a specific group of non-blocking HTTP requests. `NHttpRequestHandler` implementations are expected to take care of protocol specific aspects, whereas individual request handlers are expected 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 send back to the client in response to the given request.

```
NHttpRequestHandler myRequestHandler = new NHttpRequestHandler() {
```

```
    public ConsumingNHttpEntity entityRequest(
            HttpEntityEnclosingRequest request,
            HttpContext context) throws HttpException, IOException {
        // Buffer imcoming content in memory for simplicity
        return new BufferingNHttpEntity(request.getEntity(),
                new HeapByteBufferAllocator());
    }

    public void handle(
            HttpRequest request,
            HttpResponse response,
            NHttpResponseTrigger trigger,
            HttpContext context) throws HttpException, IOException {
        response.setStatusCode(HttpStatus.SC_OK);
        response.addHeader("Content-Type", "text/plain");
        response.setEntity(
            new NStringEntity("some important message"));
        // Submit response immediately for simplicity
        trigger.submitResponse(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.

2.8.1.2. Asynchronous response trigger

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. The worker thread can use the instance of `NHttpResponseTrigger` passed as a parameter to the `NHttpRequestHandler#handle` method to submit a response as at a later point of time once the response becomes available.

```
NHttpRequestHandler myRequestHandler = new NHttpRequestHandler() {

    public ConsumingNHttpEntity entityRequest(
            HttpEntityEnclosingRequest request,
            HttpContext context) throws HttpException, IOException {
        // Buffer imcoming content in memory for simplicity
        return new BufferingNHttpEntity(request.getEntity(),
                new HeapByteBufferAllocator());
    }

    public void handle(
            HttpRequest request,
            HttpResponse response,
            NHttpResponseTrigger trigger,
            HttpContext context)
                throws HttpException, IOException {
        new Thread() {

        @Override
        public void run() {
            try {
                Thread.sleep(10);
            }
            catch(InterruptedException ie) {}
            try {
                URI uri = new URI(request.getRequestLine().getUri());
                response.setStatusCode(HttpStatus.SC_OK);
```

```
                response.addHeader("Content-Type", "text/plain");
                response.setEntity(
                    new NStringEntity("some important message"));
                trigger.submitResponse(response);
            } catch(URISyntaxException ex) {
                trigger.handleException(
                    new HttpException("Invalid request URI: " +
                    ex.getInput()));
            }
        }

        }.start();
    }

};
```

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

2.8.1.3. 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 `NHttpRequestHandlerResolver` 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: `NHttpRequestHandlerRegistry` supports only three formats: `*`, `<uri>*` and `*<uri>`.

```
// Initialize asynchronous protocol handler
AsyncNHttpServiceHandler handler;

NHttpRequestHandlerRegistry handlerResolver =
    new NHttpRequestHandlerRegistry();
handlerReqistry.register("/service/*", myRequestHandler1);
handlerReqistry.register("*.do", myRequestHandler2);
handlerReqistry.register("*", myRequestHandler3);

handler.setHandlerResolver(handlerResolver);
```

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

2.8.2. Asynchronous HTTP client handler

`AsyncNHttpClientHandler` is a fully asynchronous HTTP client side protocol handler that implements the essential requirements of the HTTP protocol for the client side message processing as described by RFC 2616. `AsyncNHttpClientHandler` is capable of executing HTTP requests with nearly constant memory footprint for individual HTTP connections. The handler stores headers of HTTP messages in memory, while content of message bodies is streamed directly from the entity to the underlying channel (and vice versa) using `ConsumingNHttpEntity` and `ProducingNHttpEntity` interfaces.

When using this implementation, it is important to ensure that entities supplied for writing implement `ProducingNHttpEntity`. Doing so will allow the entity to be written out asynchronously. If entities supplied for writing do not implement the `ProducingNHttpEntity` interface, a delegate is added that buffers the entire contents in memory. Additionally, the buffering might take place in the I/O dispatch thread, which could cause I/O to block temporarily. For best results, one must ensure that all entities set on HTTP requests from `NHttpRequestExecutionHandler` implement `ProducingNHttpEntity`.

If incoming responses enclose a content entity, `NHttpRequestExecutionHandler` is expected to return a `ConsumingNHttpEntity` for reading the content. After the entity is finished reading the data, `NHttpRequestExecutionHandler#handleResponse()` method is called to process the response.

If incoming responses enclose a content entity, `NHttpRequestExecutionHandler` is expected to return a `ConsumingNHttpEntity` for reading the content. After the entity is finished reading the data, `NHttpRequestExecutionHandler#handleResponse()` method is called to process the response.

`AsyncNHttpClientHandler` 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 HTTP request executor is expected to take care of application specific content generation and processing.

```
//  Initialize HTTP parameters
HttpParams params;
//Initialize HTTP processor
HttpProcessor httpproc;
//Create HTTP request execution handler
NHttpRequestExecutionHandler execHandler;

AsyncNHttpClientHandler handler = new AsyncNHttpClientHandler(
         httpproc,
         execHandler,
         new DefaultConnectionReuseStrategy(),
         params);
```

2.8.2.1. Asynchronous HTTP request execution handler

Asynchronous HTTP request execution handler can be used by client-side protocol handlers to trigger the submission of a new HTTP request and the processing of an HTTP response.

HTTP request execution exents as defined by the `NHttpRequestExecutionHandler` interface:

- **`initalizeContext:`** Triggered when a new connection has been established and the HTTP context needs to be initialized. The attachment object passed to this method is the same object which was passed to the connecting I/O reactor when the connection request was made. The attachment may optionally contain some state information required in order to correctly initialize the HTTP context.

- **`submitRequest:`** Triggered when the underlying connection is ready to send a new HTTP request to the target host. This method may return null if the client is not yet ready to send a request. In this case the connection will remain open and can be activated at a later point. If the request encloses an entity, the entity must be an instance of `ProducingNHttpEntity`.

- **`responseEntity:`** Triggered when a response is received with an entity. This method should return a `ConsumingNHttpEntity` that will be used to consume the entity. Null is a valid response value, and will indicate that the entity should be silently ignored. After the entity is fully consumed, `handleResponse` method is called to notify a full response and enclosed entity are ready to be processed.

- **`handleResponse:`** Triggered when an HTTP response is ready to be processed.

- **`finalizeContext:`** Triggered when the connection is terminated. This event can be used to release objects stored in the context or perform some other kind of cleanup.

```
NHttpRequestExecutionHandler execHandler =
    new NHttpRequestExecutionHandler() {

    private final static String DONE_FLAG = "done";

    public void initalizeContext(
        HttpContext context,
        Object attachment) {
        if (attachment != null) {
            HttpHost virtualHost = (HttpHost) attachment;
            context.setAttribute(ExecutionContext.HTTP_TARGET_HOST,
                virtualHost);
        }
    }

    public void finalizeContext(HttpContext context) {
        context.removeAttribute(DONE_FLAG);
    }

    public HttpRequest submitRequest(HttpContext context) {
        // Submit HTTP GET once
        Object done = context.getAttribute(DONE_FLAG);
        if (done == null) {
            context.setAttribute(DONE_FLAG, Boolean.TRUE);
            return new BasicHttpRequest("GET", "/");
        } else {
            return null;
        }
    }

    public ConsumingNHttpEntity responseEntity(
        HttpResponse response,
        HttpContext context) throws IOException {
        // Buffer imcoming content in memory for simplicity
        return new BufferingNHttpEntity(response.getEntity(),
                new HeapByteBufferAllocator());
    }

    public void handleResponse(
        HttpResponse response,
        HttpContext context) throws IOException {
        System.out.println(response.getStatusLine());
        if (response.getEntity() != null) {
            System.out.println(
                EntityUtils.toString(response.getEntity()));
        }
    }

};
```

2.8.3. Compatibility with blocking I/O

In addition to asynchronous protocol handlers described above HttpCore ships two variants of HTTP protocol handlers that emulate blocking I/O model on top of non-blocking one and allow message content to be produced and consumed using standard `java.io.OutputStream` / `java.io.InputStream` API. Compatibility protocol handlers can work with HTTP request handlers and request executors that rely on blocking HttpEntity implementations.

Compatibility protocol handlers rely 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 executors / HTTP request processors are expected to take care of application specific content generation and processing.

2.8.3.1. Buffering protocol handlers

`BufferingHttpServiceHandler` and `BufferingHttpClientHandler` are protocol handler implementations that provide compatibility with the blocking I/O by storing the full content of HTTP messages in memory. Request / response processing callbacks fire only when the entire message content has been read into a in-memory buffer. Please note that request execution / request processing take place the main I/O thread and therefore individual HTTP request executors / request handlers must ensure they do not block indefinitely.

Buffering protocol handler should be used only when dealing with HTTP messages that are known to be limited in length.

2.8.3.2. Throttling protocol handlers

`ThrottlingHttpServiceHandler` and `ThrottlingHttpClientHandler` are protocol handler implementations that provide compatibility with the blocking I/O model by utilizing shared content buffers and a fairly small pool of worker threads. The throttling protocol handlers allocate input / output buffers of a constant length upon initialization and control the rate of I/O events in order to ensure those content buffers does not ever overflow. This helps ensure nearly constant memory footprint for HTTP connections and avoid out of memory conditions while streaming content in and out. Request / response processing callbacks fire immediately when a message is received. The throttling protocol handlers delegate the task of processing requests and generating response content to an !Executor, which is expected to perform those tasks using dedicated worker threads in order to avoid blocking the I/O thread.

Usually throttling protocol handlers need only a modest number of worker threads, much fewer than the number of concurrent connections. If the length of the message is smaller or about the size of the shared content buffer worker thread will just store content in the buffer and terminate almost immediately without blocking. The I/O dispatch thread in its turn will take care of sending out the buffered content asynchronously. The worker thread will have to block only when processing large messages and the shared buffer fills up. It is generally advisable to allocate shared buffers of a size of an average content body for optimal performance.

2.8.4. Connection event listener

Protocol handlers like the rest of HttpCore classes do not do logging in order to not impose a choice of a logging framework onto the users. However one can add logging of the most important connection events by injecting a `EventListener` implementation into the protocol handler.

Connection events as defined by the `EventListener` interface:

- **`fatalIOException`:** Triggered when an I/O error caused the connection to be terminated.
- **`fatalProtocolException`:** Triggered when an HTTP protocol error caused the connection to be terminated.
- **`connectionOpen`:** Triggered when a new connection has been established.
- **`connectionClosed`:** Triggered when the connection has been terminated.
- **`connectionTimeout`:** Triggered when the connection has timed out.

2.9. Non-blocking TLS/SSL

2.9.1. 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. Individual protocol handlers should be able to work with SSL sessions without special preconditions or modifications. However, I/O dispatchers need to take some additional actions to ensure correct functioning of the transport layer encryption.

- When the underlying I/O session has been created, the I/O dispatch must call `SSLIOSession#bind()` method in order to put the SSL session either into a client or a server mode.
- 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 nessary SSL handshaking and encrypt application data.

2.9.1.1. SSL I/O session handler

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

SSL events as defined by the `SSLIOSessionHandler` 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.

```
// Get hold of new I/O session
IOSession iosession;

// Initialize default SSL context
SSLContext sslcontext = SSLContext.getInstance("SSL");
sslcontext.init(null, null, null);

SSLIOSession sslsession = new SSLIOSession(
    iosession, sslcontext, new SSLIOSessionHandler() {

    public void initalize(
        SSLEngine sslengine,
        HttpParams params) throws SSLException {
        // Ask clients to authenticate
        sslengine.setWantClientAuth(true);
        // 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(
        SocketAddress remoteAddress,
        SSLSession session) throws SSLException {
        X509Certificate[] certs = session.getPeerCertificateChain();
        // Examine peer certificate chain
        for (X509Certificate cert: certs) {
            System.out.println(cert.toString());
        }
    }

});
```

2.9.2. SSL I/O event dispatches

HttpCore provides `SSLClientIOEventDispatch` and `SSLServerIOEventDispatch` I/O dispatch implementations that can be used to SSL enable connections managed by any arbitrary I/O reactor. The dispatches take all the necessary actions to wrap active I/O sessions with the SSL I/O session decorator and ensure correct handling of the SSL protocol handshaking.

Chapter 3. Advanced topics

3.1. 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.

3.1.1. 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 = new BasicLineParser();
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"
```

3.1.2. 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;
HttpParams params = new BasicHttpParams();
SessionInputBuffer inbuffer = new SocketInputBuffer(
    socket1, 4096, params);
SessionOutputBuffer outbuffer = new SocketOutputBuffer(
    socket2, 4096, params);

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;

HttpParams params = new BasicHttpParams();
SessionInputBuffer inbuffer = new SessionInputBufferImpl(
    4096, 1024, params);
SessionOutputBuffer outbuffer = new SessionOutputBufferImpl(
    4096, 1024, params);

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);
}
```

3.1.3. HTTP message parsers and formatter

HttpCore also provides a coarse-grained facade type of 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;

HttpParams params = new BasicHttpParams();

HttpMessageParser requestParser = new HttpRequestParser(
        inbuffer,
        new BasicLineParser(),
        new DefaultHttpRequestFactory(),
        params);

HttpRequest request = (HttpRequest) requestParser.parse();

HttpMessageWriter requestWriter = new HttpRequestWriter(
        outbuffer,
        new BasicLineFormatter(),
        params);

requestWriter.write(request);
```

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

```
SessionInputBuffer inbuffer;
SessionOutputBuffer outbuffer;

HttpParams params = new BasicHttpParams();

HttpMessageParser responseParser = new HttpResponseParser(
        inbuffer,
        new BasicLineParser(),
```

```
        new DefaultHttpResponseFactory(),
        params);

HttpResponse response = (HttpResponse) responseParser.parse();

HttpMessageWriter responseWriter = new HttpResponseWriter(
        outbuffer,
        new BasicLineFormatter(),
        params);

responseWriter.write(response);
```

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

```

SessionInputBuffer inbuffer;
SessionOutputBuffer outbuffer;

HttpParams params = new BasicHttpParams();

NHttpMessageParser requestParser = new HttpRequestParser(
        inbuffer,
        new BasicLineParser(),
        new DefaultHttpRequestFactory(),
        params);

HttpRequest request = (HttpRequest) requestParser.parse();

NHttpMessageWriter requestWriter = new HttpRequestWriter(
        outbuffer,
        new BasicLineFormatter(),
        params);

requestWriter.write(request);
```

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

```

SessionInputBuffer inbuffer;
SessionOutputBuffer outbuffer;

HttpParams params = new BasicHttpParams();

NHttpMessageParser responseParser = new HttpResponseParser(
        inbuffer,
        new BasicLineParser(),
        new DefaultHttpResponseFactory(),
        params);

HttpResponse response = (HttpResponse) responseParser.parse();

NHttpMessageWriter responseWriter = new HttpResponseWriter(
        outbuffer,
        new BasicLineFormatter(),
        params);

responseWriter.write(response);
```

3.1.4. 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.

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

3.2. Customizing HTTP connections

One can customize the way HTTP connections parse and format HTTP messages by extending the default implementations and overriding factory methods and replacing the default parser or formatter implementations with a custom one.

For blocking HTTP connections one also can provide custom implementation of session input/output buffers.

```
class MyDefaultNHttpClientConnection
                        extends DefaultNHttpClientConnection {

    public MyDefaultNHttpClientConnection(
            IOSession session,
            HttpResponseFactory responseFactory,
            ByteBufferAllocator allocator,
            HttpParams params) {
        super(session, responseFactory, allocator, params);
    }

    @Override
    protected NHttpMessageWriter createRequestWriter(
            SessionOutputBuffer buffer,
            HttpParams params) {
        return new HttpRequestWriter(
            buffer, new BasicLineFormatter(), params);
    }

    @Override
    protected NHttpMessageParser createResponseParser(
            SessionInputBuffer buffer,
            HttpResponseFactory responseFactory,
            HttpParams params) {
        return new HttpResponseParser(
            buffer, new BasicLineParser(), responseFactory, params);
    }

};
```

For non-blocking HTTP connection implementation one can replace the default HTTP message parser and formatter implementations. The session input/output buffer implementations can be overridden at the I/O reactor level.

```
class MyDefaultHttpClientConnection
                        extends DefaultHttpClientConnection {

    @Override
    protected SessionInputBuffer createSessionInputBuffer(
            Socket socket,
            int buffersize,
            HttpParams params) throws IOException {
        return new MySocketInputBuffer(socket, buffersize, params);
    }
```

```
    @Override
    protected SessionOutputBuffer createSessionOutputBuffer(
            Socket socket,
            int buffersize,
            HttpParams params) throws IOException {
        return new MySocketOutputBuffer(socket, buffersize, params);
    }

    @Override
    protected HttpMessageWriter createRequestWriter(
            SessionOutputBuffer buffer,
            HttpParams params) {
        return new MyHttpRequestWriter(
            buffer, new BasicLineFormatter(), params);
    }

    @Override
    protected HttpMessageParser createResponseParser(
            SessionInputBuffer buffer,
            HttpResponseFactory responseFactory,
            HttpParams params) {
        return new MyHttpResponseParser(
        buffer, new BasicLineParser(), responseFactory, params);
    }

};
```