GuidedTourOfHttpCore

A Guided Tour of HttpCore (module-main)

Welcome

Welcome, visitors, to this guided tour of HttpCore. I am your tour guide. If you could please come a little closer and gather around me, so I don't have to shout? Thank you, that's much better. I'm about to give you a short introduction, and then we'll visit some interesting places in HttpCore so you can see how it works. Whenever you've got a question, feel free to ask. That's what I'm here for, to answer your questions.

As you probably know, HTTP is a protocol for exchanging messages between a client and a server. It's in widespread use, and it typically is running on top of plain TCP/IP or secure TLS/SSL sockets.

Here at Apache, there is an implementation of the client side of that protocol called the Commons HttpClient. Informally, we also call it "the 3.x codebase" or simply "the old code". It proved quite useful to a lot of people, but the old code has severe limitations in its design. For example, there is a class called `HttpMethodBase`. It represents a request and a response at the same time, and it also implements logic for processing both. This kind of design where different things are crammed together in a single place makes it really hard to maintain or extend the code.

Therefore we started a new successor project called HttpComponents. Based on the experience gained with the old code, it implements the HTTP protocol with a new approach. Above all, there are several modules dealing with different aspects of the big problem. As you can gather from its name, the HttpCore module is at the very heart of this effort. It defines stuff on which all the other modules depend and rely.

HttpCore deals with representation of HTTP messages, and with transport logic for sending and receiving those messages. It also defines a kind of framework infrastructure so other modules can plug in functionality. Unlike the old code, HttpCore is not specific to the client side of HTTP communication, it can also be used for the server side. And because it is so fundamentally different from its predecessor, we put all the code into an all-new package hierarchy so you don't confuse them.

Q: I have a question.

A: Yes, please? What would you like to ask?

Q:
If it is in a new package hierarchy, applications written
for the old code will not be able to use the new one?

A:
Yes, that is correct. Because the old code was limited in its design,
we had to change the API. Applications have to be rewritten to
make use of the new code. There was no way to avoid this.
The all-new package names at least make sure that both old and new code
can be used in the same environment, for example a Servlet engine,
without interference.

Now, if you would like to follow me to the main hall... it's called package `org.apache.http`. You may want to keep the JavaDocs at hand, that will make it easier for you to follow my explanations.

Messages

The first problem we had to deal with is the representation of messages. If you don't know how to represent a message, you can't send or receive it, right? So here we have a set of interfaces for the building blocks of an HTTP message. There's the `RequestLine` for a request and the `StatusLine` for a response, both containing a `ProtocolVersion`. The latter is so elementary that we made it a class instead of an interface, and of course we have the `HttpVersion` derived from it. Then we have a `Header` with name and value, where the value can have multiple `HeaderElement}}s. And finally there is the message body, the {{HttpEntity`.

Q:
`HttpVersion` derived from `ProtocolVersion`?
Wouldn't the protocol always be HTTP in HttpCore?

A:
Not quite. There is at least one other protocol,
the Session Initiation Protocol SIP,
which has a message format identical to that of HTTP.
Only the protocol name and version differs.
Since it's so similar, we tried to keep the door open.

Q: That makes sense.

So, out of these building blocks, we collect messages. Every `HttpMessage` has headers, which can be added or deleted at will.
`HttpRequest` adds the request line,
`HttpEntityEnclosingRequest` an entity.
`HttpResponse` adds a status line and also an entity. For convenient integration into frameworks that explore the Factory pattern, there are factory interfaces for both requests and responses.

Q: Aren't there responses without an entity?

A:
Yes. You are very attentive!
You don't have to provide an entity to the response, you can leave it null.
With requests, you usually know in advance whether you want to provide an entity or not. Also, many requests are GET requests and don't have an entity.
That's why we created two different interfaces.
Responses mostly have an entity, except in very special cases. Having to know in advance whether there will be an entity or not would have made the API cumbersome to use in some situations, so we went with a single interface.

Q:
You said headers can be added and deleted from an `HttpMessage`.
What if I want a read-only message?

A:
We've given up the idea of distinguishing between modifiable and non-modifiable messages. It created an insane number of interfaces, and we had to spread instanceof checks and downcasts all over the place.
HttpCore after all is meant for people who know what they do.
If you want a message to remain unchanged, simply don't change it 😉
If you really have to prevent modifications, you can implement the interface with a custom class that throws an exception whenever a modifying method is called.

Now, if you would come over here and have a look through this window into the adjoining room? It's a bit too small for all of us to go into, but you can see the important things from here. It is called package `org.apache.http.message`. Notice that there are basic implementations for all of the message representation interfaces. You'll hardly need more than those when writing an application that uses HttpCore directly.

Q:
I can't see implementations for `GetRequest` and `PostRequest` and so on?

A:
You are right, we don't have those convenience classes in core.
Core is hardcore. If you want a GET request, you just create a basic request and pass "GET" as the HTTP method name. Likewise for POST or PUT, except you'd create a basic entity enclosing request for those.
You'll find convenience classes for the default HTTP methods in the client, but they are really superfluous in the core.

Q:
But then I could create an entity enclosing request with GET as the method name. That doesn't make sense, GET requests never have an entity!?

A:
Yes, you can do all sorts of stupid things with core.
Core is hardcore, and meant for people who know what they're doing.
And maybe you really want to create a GET request with an entity, for example to test how a server responds to invalid requests?

Q:
An interesting example. I hadn't thought about that.

All right, are there more questions about the basic implementations? No? Good, then let's move on to the room over there. It is called package `org.apache.http.entity`. You can find a selection of message entities in there. Message entities are really not that much different from the request entities in HttpClient 3.x, except they are no longer tied to the client side. As in the old code, there are entities getting their content from a string, byte array, file, or input stream. The `BasicHttpEntity` is what we use when a message is received over a connection. You'll see the connections later today. We also have some advanced stuff for wrapping and buffering entities, and an `EntityTemplate` that simplifies writing a new entity if you have to.

Q:
There used to be a multipart entity in the old code?

A:
Indeed, there is. But that didn't make it into core.
It was considered slightly out of scope even for the old code.
A replacement is in module-httpmime of HttpClient.
We are using mime4j there,
which drags in a few more dependencies.
HttpCore has no external dependencies at all.

Any more questions about entities? Fine, then let's pass through this door, back into the main hall. We're going to have a look at connections next.

Connections

Connections are needed to send HTTP message from client to server or the other way 'round. On the interface level, we have the `HttpConnection`. It allows for checking whether a connection is open, for closing it or shutting it down, and for getting statistical data if such has been gathered. To actually send and receive messages, you have to use either `HttpClientConnection` or `HttpServerConnection`, depending on what you implement. Obviously, the client connection allows for sending requests and receiving responses, whereas the server connections receives requests and sends responses. Messages are passed to and from the connections in terms of the interfaces we have just seen. We require two calls for sending the message header and the message entity. That allows for explicit handling of the expect-continue handshake, for example.

Q:
I don't see a method to open a connection?

A:
You have an eagle's eyes, don't you? That is absolutely correct,
there is no method for opening a connection in the interface.
It took a good deal of discussions until we got to that point.
Opening a connection can be trickier than it might seem at first
glance, so we left it out of the core API.

Q:
So how do I use a connection if I can't open it?

A:
Well, the API is different from the implementation. There is no
`open()` in the API, but your implementation can offer that method.
The default implementations we ship in core expect to be given
an open socket, which you can create in any way you want.

Q:
Talking of sockets, I don't see a socket in the interfaces either.
Wouldn't it be useful, for example to configure TCP/IP settings?

A:
Oh yes, it's definitely useful to know the socket. But, you see,
it doesn't belong into the core API. Somebody might want to use the
API with some native communication library instead of Java sockets.
But we'll have a look at the default implementations right away,
you'll see the socket there.

Q:
Does that mean one has to downcast to an implementation class in order
to obtain the IP address and port number connected to?

A:
Oh no, it's not *that* bad. You see, here is one more interface
`HttpInetConnection` that provides access to IP addresses and
port numbers, both local and remote. It's an optional interface, but
it's supported by all our default implementations. You only have to
cast to the interface, not to an implementation class.

Before you ask any more questions, it's probably best we move on into the next room, which is called package `org.apache.http.impl`. As you can see, there is a whole bunch of connection implementation classes. Don't let that confuse you, it's just for keeping the code maintainable. All you really need to look at are the two classes `DefaultHttpClientConnection` and `DefaultHttpServerConnection`. You see, there are the `bind` operations I told you about, where you pass in an open socket to have an open connection. And inherited from a base class, there also is a `getSocket` method.

Q:
There are a lot of inherited methods. What's this serializer stuff?
And the data transmitter?

A:
Don't worry about those. We provide reasonable defaults.
It'll all just work by itself, you don't have to do anything.
It's a kind of magic 😉

Q:
There are connection re-use strategies here, and I've seen an interface in the main hall. That sounds like connection management?

A:
Yes, almost, but not quite. There is no connection management in core. But there are cases where core has to decide about closing a connection. Remember, `close()` is in the generic interface. The re-use strategies are used to decide about closing connections.

Q:
Can those re-use strategies query the statistical information you've mentioned before?

A:
Yes, that's the idea. A re-use strategy can look at the headers of request or response, but also at the statistical data of the connection if that is available.

Q: Where do these doors lead to?

A:
Eh, please don't go there, thank you. The adjoining rooms `org.apache.http.impl.entity` and `...impl.io` are where the transport encodings are handled.
You know, the magic stuff I mentioned.

Now please, visitors... I know that the connections look very interesting and complicated, but you really don't want to miss the exciting things still coming up. So, if you follow me back to the main hall, and then on to the next room...

Execution

Here we are in package `org.apache.http.protocol`. This is the home of the framework for executing the higher levels of HTTP. Remember that the lower levels, in particular transport encodings, are dealt with automagically by the connections we have just left. The protocol framework here is concerned with putting the appropriate headers into messages, and with calling the connection methods at the right time in the right sequence.

For example, the expect-continue handshake is dealt with here, both on the client and server side. For those of you that are not familiar with the details of that handshake, I'll explain it briefly. When sending a message with a body that is large or tricky to generate, clients don't want to risk sending the message data just to get a simple error response from the server, for example because authentication is required. In that case, the client will put a special Expect: header into the request and send only the message headers. The server is expected to check the message headers, and to respond with a status code of 100 if it finds everything ready for processing the request entity. Only then will the client send the rest of the request. If the server detects a problem, it responds with the appropriate error status code and the request body is never sent.

Here we have the `HttpRequestExecutor`, the client side implementation for protocol execution. It handles the expect-continue handshake, and it also checks whether an incoming response has an entity that needs to be read. For the server side we have `HttpService`, which checks whether the incoming request has an entity, and uses `HttpExpectationVerifier` if the expect-continue handshake is employed. Both use an `HttpProcessor` to modify and interpret headers.

The framework for setting and interpreting headers is based on ***interceptors***. Those are little classes which take care of one specific aspect, often just a single header. These are collected into a list of interceptors that need to be executed on a message before it is sent, or after it is received. A range of typically needed interceptors is provided, I'll just pick some examples.

Here we have the `RequestUserAgent`. It is a request interceptor for outgoing requests, so it is executed on requests on the client side before they are sent. Its only task is to add a User-Agent header, if there is none in the request. If you don't want a User-Agent header to be sent, you just don't add this interceptor to your list.

Q:
How does `RequestUserAgent` know the value for the header?

A:
A very good question. We are keeping parameters with the request.
Those will be the next station of this guided tour.

A trickier interceptor is `RequestContent`, also applied before a request is sent on the client side. It checks whether there is an entity in the request and sets up Content-Length and Transfer-Encoding headers if so. This is a must-have interceptor if you want to send a request entity. On the server side, `ResponseContent` does the same for the response.

Q:
Didn't you say that transfer encodings are handled automagically?

A:
Yes, I did. These interceptors are the wizards that make it all happen.

The already mentioned `HttpProcessor` holds lists of request and response interceptors that should be applied. You set it up once when your application initializes.

Q:
There are very many interceptors here. How do I know which ones I need?

A:
That is a tricky thing. You should stick to the interceptor lists used
in the examples. If that doesn't do what you want, just ask by posting
your question to the user
mailing list.

Q:
If I need to authenticate a request, I would use an interceptor that
asks the user for the password?

A:
NO! Ahem, sorry. No. You should never execute a blocking operation of
this kind in an interceptor, and in particular not user interactions.
In general, you don't know what kind of background process will execute
the interceptors. You could stall the whole application, or even others
if it is running in a shared environment.

Q:
Then an interceptor that asks the user to confirm cookies is also
not a good idea? But how should I do it?

A:
You should perform user interaction before or after the execution of the
interceptors. For authentication, you would ask for the password before
executing the request, and then give the password to the interceptor.
For cookies, you take an interceptor that puts the cookies in a separate
location, and ask for confirmation when the request execution is done.
I was just about to explain how applications interact with interceptors.

You see this interface here, `HttpContext`. That is a collection of named attributes, where names are strings and attributes can be any kind of Java object. When a request is executed, it has one specific context. Likewise when a request is being serviced on the server side, of course. The interceptors, and many other parts of the framework, have access to this context. So your application can put some data - like a password - into the context, and an interceptor picks it up. On the other hand, an interceptor can put data into the context - like incoming cookies - and your application picks that up after the execution.

The context is also the place to keep session information, like the cookies that should be sent or passwords that have already been entered. Mind you, core does not handle cookies or authentication. Core is hardcore, it just provides the framework for doing that. The examples show what attributes need to be present in the context for the default interceptors to work. We have synchronized and unsynchronized implementations of the `HttpContext` interface.

Now, if you would kindly follow me to the last stop on our little tour...

Parameters

This is package `org.apache.http.params`, home of the parameter framework. We've introduced the preferences framework with version 3.0 of the old code. The 4.0 version is a natural evolution of that rather than a radical redesign. We keep maps of named parameters in instances of `HttpParams`. Parameters get attached to HTTP messages, so they are available to all objects involved in processing a message: interceptors, connections, and whatever else other modules are going to add on top of core.

The names of parameters are defined in `PNames` interfaces, where each interface lists parameters for a particular part of the framework. We also have `Bear` classes for these parameter sets. These beans don't store the parameters in attributes, but put them into a parameter map. This comes in handy if you want to use something like the Spring framework, which can populate beans from configuration files but wouldn't know what to do with a map.

In the old code, parameters were hierarchical. This feature is still present, we can link a map of parameters with another one providing defaults. However, this feature should *never* be used by applications directly. Parameters may and will be linked inside the framework, and having both application and framework set up parameter hierarchies would wreak havoc on both.

Q:
What's the difference between parameters and contexts?
Both are maps of named attributes.

A:
From a framework perspective, parameters are read-only. The application
prepares parameters in advance, then the frameworks reads them.
The context is updated by the framework.
Furthermore, parameters are meant to hold data, whereas the context can
hold any kind of attribute. As a rule of thumb, if it is something you'd
write into a properties file, that's a candidate for parameters. If you
need to set up a callback at runtime, that goes into the context.

Caution has to be used when updating parameters after they have been passed to the framework. You should avoid to update a parameter set at all while execution or servicing is in progress. The default implementation of `HttpParams` is unsynchronized, because the framework will use it read-only.

The parameter *values* themselves should be read-only at all times. So if for example you stored a modifiable map as a parameter value, never modify that map again. If you have to update the parameter set with a new map, then copy the old one, modify the copy, and replace the old value with the modified copy.

Q:
If parameters are read-only for the framework, why is the interface read-write?

A:
Good question! That's for the users,
so they can get and modify parameters without typecasting:

```
request.getParams().setParameter("name", value);
```

Q:
What's with these `HttpProtocolParams` and `HttpConnectionParams`?
Are these special implementations of the interface?

A:
Ah, no. Those classes contain static helper methods for
getting and setting the respective parameters.
This encapsulates typecasts and provides at least some type safety:

```
HttpProtocolParams.setVersion(request.getParams(), HttpVersion.HTTP_1_0);
HttpVersion version = HttpProtocolParams.getVersion(request.getParams());
```

Q:
Is there a helper that loads parameters from a properties file?

A:
No, core is hardcore. We don't deal with configuration via properties files
in core. If we supported properties files today, we'd be asked to support
XML configuration tomorrow, and something else the day after.
There would be no end to it.
Besides, instantiating parameters with the correct type is not trivial.
A string parameter needs to be stored as `String`, whereas an integer
parameter needs to be stored as `Integer`. It becomes even worse for
custom parameter types. We don't want this kind of type conversion logic
in core. Maybe in an extra module, sometime.

Farewell

I hope you enjoyed our tour of the HttpCore module and found the experience enlightening. If you have any more questions, do not hesitate to post them on the user mailing list. Saying that, you might want to search the archives of the mailing lists first, in case somebody else already got an answer to a similar question. We are also considering to offer guided tours of other modules in the future. We'd be happy if you join one of those when they become available.

Thank you all, and see you next time!