HttpClientConfiguration

HttpClient configuration and prefernece API

HttpClient customization

HttpClient instances can be created either using HttpClientBuilder or factory methods of the HttpClients utility class.

This code snippet shows how to create HttpClient instance with default configuration. The instance will be configured to use a pool of connections with maximum two concurrent connections for the same route (host).

```
CloseableHttpClient client = HttpClients.createDefault();
```

This code snippet shows how to create an HttpClient instance based on system properties. The instance will be configured to use a pool of connections. The following system properties will be taken into account:

- ssl.TrustManagerFactory.algorithm
- javax.net.ssl.trustStoreType
- javax.net.ssl.trustStore
- javax.net.ssl.trustStoreProvider
- javax.net.ssl.trustStorePassword
- java.home
- ssl.KeyManagerFactory.algorithm
- javax.net.ssl.keyStoreType
- javax.net.ssl.keyStore
- javax.net.ssl.keyStoreProvider
- javax.net.ssl.keyStorePassword
- http.proxyHost
- http.proxyPort
- http.nonProxyHosts
- http.keepAlive
- http.maxConnections
- http.agent

```
CloseableHttpClient client = HttpClients.createSystem();
```

This code snippet shows how to create an HttpClient instance with a minimal configuration. This instance will be configured to use a pool of connections with maximum two concurrent connections for the same route (host). The only request level configuration parameters that the minimal HttpClient takes into account are timeouts (socket, connect, and connection request). All other request parameters will have no effect on request execution.

```
CloseableHttpClient client = HttpClients.createMinimal();
```

Please note that HttpClient instances created with HttpClientBuilder or HttpClients are immutable. Their configuration can no longer be altered.

This code snippet shows how to create an HttpClient instance with a custom configuration. One can disable certain protocol aspects such as automatic redirect handling to have them completely removed from the request execution chain and make request execution more efficient.

```
CloseableHttpClient httpclient = HttpClients.custom()
    .disableAutomaticRetries()
    .disableConnectionState()
    .disableContentCompression()
    .disableRedirectHandling()
    .useSystemProperties()
    .build();
```

One can choose to configure HttpClient to use system properties and then explicitly override only specific aspects through custom configuration.

```
CloseableHttpClient httpclient = HttpClients.custom()
    .useSystemProperties()
    .setProxy(new HttpHost("myproxy", 8080))
    .build();
```

Request configuration

This code snippet shows how to create custom request configuration.

```
RequestConfig defaultRequestConfig = RequestConfig.custom()
    .setSocketTimeout(5000)
    .setConnectTimeout(5000)
    .setConnectionRequestTimeout(5000)
    .setStaleConnectionCheckEnabled(true)
    .build();
```

Request configuration can either be set at the client level as defaults for all requests without explicit configuration or at the request level.

```
CloseableHttpClient httpclient = HttpClients.custom()
    .setDefaultRequestConfig(defaultRequestConfig)
    .build();
```

Please note that requests do not automatically inherit client level request configuration, if it overridden at the request level. Configuration defaults must be explicitly copied.y

```
HttpGet httpget = new HttpGet("http://www.apache.org/");
RequestConfig requestConfig = RequestConfig.copy(defaultRequestConfig)
    .setProxy(new HttpHost("myotherproxy", 8080))
    .build();
httpget.setConfig(requestConfig);
```

Request execution context

HTTP protocol processors used internall by HttpClient are state-less, that is, they maintain no conversational between individual HTTP exchanges. However, conversational state such as HTTP cookies or authentication details can be preserved in the execution context represented by HttpClientContext class.

The execution context can be shared by multiple HTTP exchanges if they belong to the same logical HTTP session. The context can be set up with a particular state prior to executing the HTTP session. The context will also be updated in the course of the session. The conversational state can be examined and updated after each individual HTTP exchange.

Default settings and configuration at the client level will be automatically added to the execution context if not explicitly set in the context by the user.

```
CloseableHttpClient httpclient = HttpClients.custom()
    .setDefaultCookieStore(defaultCookieStore)
    .setDefaultCredentialsProvider(defaultCredentialsProvider)
    .setDefaultRequestConfig(defaultRequestConfig)
    .build();

HttpClientContext context = HttpClientContext.create();
context.setCookieStore(customCookieStore);
context.setCredentialsProvider(customCredentialsProvider);
context.setRequestConfig(customRequestConfig)

HttpGet httpget = new HttpGet("http://www.apache.org/");
CloseableHttpResponse response = httpclient.execute(httpget, context);
try {
    // Last executed request
    context.getRequest();
    // Execution route
    context.getHttpRoute();
    // Target auth state
    context.getTargetAuthState();
    // Proxy auth state
    context.getTargetAuthState();
    // Cookie origin
    context.getCookieOrigin();
    // Cookie spec used
    context.getCookieSpec();
    // User security token
    context.getUserToken();

} finally {
    response.close();
}
```

Connection management and configuration

This code snippet shows how to create a HttpClientConnectionManager instance that keeps a pool of re-usable persistent connections. By default the connection manager will allow no more than 2 concurrent connections for the same route and no more than 20 connections in total.

```
PoolingHttpClientConnectionManager connManager = new PoolingHttpClientConnectionManager();
```

These limits can be customized if desired. One can also specify a different maximum limit specifically for a particular host.

```
connManager.setMaxTotal(100);
connManager.setDefaultMaxPerRoute(10);
connManager.setMaxPerRoute(new HttpRoute(new HttpHost("somehost", 80)), 20);
```

The PoolingHttpClientConnectionManager class can apply different configuration parameters to network sockets and HTTP connections. Socket and connection configuration can be set as defaults or applied to a specific host.

This code snippet shows how to set socket configuration.

```
SocketConfig defaultSocketConfig = SocketConfig.custom()
    .setTcpNoDelay(true)
    .build();
SocketConfig socketConfig = SocketConfig.custom()
    .setTcpNoDelay(true)
    .setSoKeepAlive(true)
    .setSoReuseAddress(true)
    .build();

connManager.setDefaultSocketConfig(defaultSocketConfig);
connManager.setSocketConfig(new HttpHost("somehost", 80), socketConfig);
```

This code snippet shows how to set socket configuration.

```
MessageConstraints messageConstraints = MessageConstraints.custom()
    .setMaxHeaderCount(200)
    .setMaxLineLength(2000)
    .build();
ConnectionConfig defaultConnectionConfig = ConnectionConfig.custom()
    .setMessageConstraints(messageConstraints)
    .build();
ConnectionConfig connectionConfig = ConnectionConfig.custom()
    .setMessageConstraints(messageConstraints)
    .setMalformedInputAction(CodingErrorAction.IGNORE)
    .setUnmappableInputAction(CodingErrorAction.IGNORE)
    .setCharset(Consts.UTF_8)
    .build();
connManager.setDefaultConnectionConfig(defaultConnectionConfig);
connManager.setConnectionConfig(new HttpHost("somehost", 80), connectionConfig);
```

A custom connection factory can also be used to customize the process of initialization of outgoing HTTP connections. Beside standard connection configuration parameters HTTP connection factory can control the size of input / output buffers as well as determine message parser / writer routines to be employed by individual HTTP connections. Custom message parser / message writer are responsible for marshaling / un-marshaling of HTTP messages transferred over the HTTP connection. There are situations when one may want to employ a more lenient parsing when dealing with broken or non-compliant server side scripts.

```
HttpMessageParserFactory<HttpResponse> responseParserFactory = new DefaultHttpResponseParserFactory() {

    @Override
    public HttpMessageParser<HttpResponse> create(
        SessionInputBuffer buffer, MessageConstraints constraints) {
        LineParser lineParser = new BasicLineParser() {

            @Override
            public Header parseHeader(CharArrayBuffer buffer) {
                try {
                    return super.parseHeader(buffer);
                } catch (ParseException ex) {
                    return new BasicHeader(buffer.toString(), null);
                }
            }

        };
        return new DefaultHttpResponseParser(
            buffer, lineParser, DefaultHttpResponseFactory.INSTANCE, constraints) {

            @Override
            protected boolean reject(CharArrayBuffer line, int count) {
                // try to ignore all garbage preceding a status line infinitely
                return false;
            }

        };
    }

};
HttpMessageWriterFactory<HttpRequest> requestWriterFactory = new DefaultHttpRequestWriterFactory();
HttpConnectionFactory<SocketClientConnection> connFactory = new DefaultClientConnectionFactory(
        8 * 1024, requestWriterFactory, responseParserFactory);
PoolingHttpClientConnectionManager connManager = new PoolingHttpClientConnectionManager(connFactory);
```

Client HTTP connection objects when fully initialized can be bound to an arbitrary network socket. The process of network socket initialization, its connection to a remote address and binding to a local one is controlled by a connection socket factory. It is generally recommended to provide a connection socket factory for SSL connections with a custom configuration as generally security requirements tend to be application specific.

SSL context for secure connections can be created either based on system or application specific properties.

```
SSLContext sslcontext = SSLSocketFactory.createSystemSSLContext();
```

One can also choose a custom hostname verifier to customize the process of hostname verification.

```
X509HostnameVerifier hostnameVerifier = new BrowserCompatHostnameVerifier();
```

This code snippet shows how to create a registry of custom connection socket factories for supported protocol schemes.

```
Registry<ConnectionSocketFactory> socketFactoryRegistry = RegistryBuilder.<ConnectionSocketFactory>create()
    .register("http", PlainSocketFactory.INSTANCE)
    .register("https", new SSLSocketFactory(sslcontext, hostnameVerifier))
    .build();
```

One can use a custom DNS resolver to override the system DNS resolution.

```
DnsResolver dnsResolver = new SystemDefaultDnsResolver() {

    @Override
    public InetAddress[] resolve(final String host) throws UnknownHostException {
        if (host.equalsIgnoreCase("myhost")) {
            return new InetAddress[] { InetAddress.getByAddress(new byte[] {127, 0, 0, 1}) };
        } else {
            return super.resolve(host);
        }
    }

};
```

This code snippet shows how to put together a pooling HttpClientConnectionManager with custom connection factory, socket factories and DNS resolver

```
PoolingHttpClientConnectionManager connManager = new PoolingHttpClientConnectionManager(
    socketFactoryRegistry, connFactory, dnsResolver);
CloseableHttpClient httpclient = HttpClients.custom()
    .setConnectionManager(connManager)
    .build();
```

Custom connection socket factories can also be provided at the request level through a local execution context. This will cause HttpClient to override the default socket initialization routines with those specified in the execution context.

```
HttpClientContext context = HttpClientContext.create();
context.setSocketFactoryRegistry(socketFactoryRegistry);
```

Please note that if a custom HTTP connection is kept alive after the request execution it may be pooled and re-used for execution of other requests.