Developer notes and secrets

I can't run this on PHP 5.3 or PHP 5.4 or ...

And you shouldn't. UPGRADE NOW!

There are still tests running for very low releases (down to at least PHP 5.4) but there is no longer any guarantee that this will work. As we work with such things as short hand arrays and chained requests, everything below v5.4 - and even 5.6 - is obsolete. The highest wish for netcurl is that we also leave all PHP releases where type hinting is incompatible, so it will probably be something for netcurl 6.2 och 7.0.

Generally having problems with PHP 5?

Yes. **For netcurl 6.1**, it's no longer my problem. It's yours and your inability to move forward with the world. Go PHP 7 NOW, so we can drop PHP 5 entirely! Netcurl still lives in an obsolete world of syntax, thanks to the slow upgrade rate. But as everybody - except you - are moving forward now, you will be left alone with your old releases.

Optionize netcurl

curl has been a big part of this library so most of the content are built around this driver, even if it's not needed. But a really huge part och at least WrapperConfig is using settings that is based on the constants. When other libraries requires special settings (i.e such that are not injected by you) the configurator could transform those settings to its right environment - or just keeps them as is inside the curlopts array.

CURLOPTs

CURLOPT options are no longer primarily set as constants, as we sometimes meet curlopts not set in the current release of curl on server level. This is very rare though, but if this happens it's a sign that the server owner runs something that is way too old for a production. In such cases, you should upgrade your system before bugtracking the library. However, for the sake, netcurl includes its own version of curlopts in case of any missing constants and the are fetched dynamically only when necessary and the default constants are missing - instead of running into undefined data warnings. If this happens it's more likely that the developer, that utilizes netcurl, has pushed in something wrong (usually something that is too old comparing to upgraded software).

The transformation happens here, in WrapperConfig:

```
    private function getCurlConstants($curlOptConstant)
    {
        if (is_array($curlOptConstant)) {
            foreach ($curlOptConstant as $curlOptKey => $curlOptValue) {
                $constantValue = @constant($curlOptKey);
                if (empty($constantValue)) {
                    // Fall back to internally stored constants if curl is not there.
                    $constantValue = @constant('TorneLIB\Module\Config\WrapperCurlOpt::NETCURL_' . $curlOptKey);
                }
                $this->options[$constantValue] = $curlOptValue;
            }
        }
    }
```


curlopt defaults in v6.0

In 6.0 some default curl constants is not in use. Information can be found at Known issues and fixes.

How we handle SSL

The below text is documented from prior releases of NetCurl 6.0 - it is a well documented fact that CURLOPT_SSL_VERIFYHOST has changed over time. In netcurl 6.1 it is still not decided whether this should be kept or discontinued to use as this was a problem discovered in a very specific version of PHP 5.4 combined with libcurl. Running this old PHP releases **should** be considered extremely disencouraged (and stupid) in a security point of view.

```
From libcurl 7.28.1 CURLOPT_SSL_VERIFYHOST is deprecated. However, using the value 1 can be used
as of PHP 5.4.11, where the deprecation notices was added. The deprecation has started before libcurl
7.28.1 (this was discovered on a server that was running PHP 5.5 and libcurl-7.22). In full debug
even libcurl-7.22 was generating this message, so from PHP 5.4.11 we are now enforcing the value 2
for CURLOPT_SSL_VERIFYHOST instead. The reason of why we are using the value 1 before this version
is actually a lazy thing, as we don't want to break anything that might be unsupported before this version.
```

SSL Certificate problems

This section covers:

- **SSL3_GET_SERVER_CERTIFICATE**
- **CURLE_SSL_CACERT**
- **SSL2_SET_CERTIFICATE (error)**

Documented in **NETCURL-13** - Getting issue details... STATUS

In some versions of PHP SSL verification fails with routines:SSL3_GET_SERVER_CERTIFICATE:certificate. For the tests, where the importance of result is not focused on SSL, we could disable the verification checks if we want to do so. In Bitbucket Pipelines docker environments errors has been discovered on some PHP releases, which we'd like to primary disable.

In version 6.0.20 a self adjusting feature was added to handle verification errors automatically. Especially the error codes **14090086** (*routines: ssl3_get_server_certificate:certificate*) and **1407E086** (*routines:SSL2_SET_CERTIFICATE*) was added to the core to make sure - if it was allowed by the system - such problems could be bypassed. By means, in this case it is equal to a security layer removal (by simply disable SSL verifications on fly).

For v6.1, netcurl is set to guess the SSL version used on the remote:

WrapperConfig

```
'CURLOPT_SSLVERSION' => CURL_SSLVERSION_DEFAULT,
```

Configuration classes and verbosity

netcurl is written so that no preconfiguration is necessary; the defaults has from 6.0 been set to be as verbose as possible. This is from v6.1 not entirely true. In 6.0 the verbosity has been used to retrieve and handle full head and body and return them to the user as is. It is true that the content return will still be intact, but the structure of how the head is returned to clients has been changed. Instead of downloading the header and transform it to fetchable content, netcurl is instead using native functions to extract the header. This is done with the CURLOPT_HEADERFUNCTION option flag. Changing this is prevented from the core parts of netcurl, even if it's still potentially available from the getCurlHandle()-method.

Most of the options available in curl is configurable, by the WrapperConfig-class, except for the settings below (not getCurlHandle included).

curlopt	static value	notes
CURLOPT_URL	**User defined url.**	Legacy for fetching an URL, that potentially makes it possible to fetch an URL from a server that is normally not the one pointed out in DNS entries. *In the initial releases of netcurl 6.0 the url was called directly from the execution parts.*
CURLOPT_HEADER	false	
CURLOPT_RETURNTRANSFER	true	
CURLOPT_AUTOREFERER	true	
CURLINFO_HEADER_OUT	true	

CURLOPT_HEADERFUNCTION	**Internal getCurlHeaderRow.**	

As for externally configurable, a developer may use something like this to set up own properties in netcurl:

```
        $curlRequest =
            (new CurlWrapper())
                ->setConfig((new WrapperConfig())->setOptions([CURLOPT_USERAGENT => 'ExternalClientName']))
                ->request(
                    sprintf('https://ipv4.netcurl.org/?func=%s', __FUNCTION__)
                );
```

... or a less complex request:

```
        $wrapperConfig = new WrapperConfig();
        $wrapperConfig->setOptions([CURLOPT_USERAGENT => 'ExternalClientName']);
        $curlWrapper = new CurlWrapper();
        $curlWrapper->setConfig($wrapperConfig);
        $curlWrapper->request(sprintf('https://ipv4.netcurl.org/?func=%s', __FUNCTION__));
```

MODULE_CURL is not completed yet, so this is examples used when the curlwrapper is called directly. More information will come soon.

Unspoofable Client & Self configuration

For v6.0 there's been a spoof protection placed in the core code. In all versions before 6.0.25, netcurl prevented its usage from "abuse" in the form of User-Agent hijacking. When a User-Agent has been set within netcurl, netcurl has forced core components to be logged in the user agent together with the user defined agent name. As of 6.0.25, as mentioned, a spoofable flag has been added to the setUserAgent parts so the user defined string is the only visible in the http requests. As of 6.1.0, this protection is entirely removed and the core curlops are set entirely free with the help of configuration class.

Global useragent handling

Normally user-agent setups is being done via internal WrapperConfig setups. However, this can be done via a global static variable in the WrapperConfig too. By using WrapperConfig::setSignature(), you can push a user agent on a global level without the need of an instance. This value has higher priority (currently at least) than the internal value. This simplifies client identification.

However, to make it harder to abuse this, it can also be disabled like this:

```
define('WRAPPERCONFIG_NO_SIGNATURE', true);
```

"Identifiers"

NETCURL-289 - Getting issue details... STATUS simplifies the ability to send sanitized useragent strings with automatic appending of driver data.
For example, if you run a client that identifies itself as "CurlyChrome", setUserAgent() can be us as usually. However, if you need to send more information about what's used internally in netcurl, you can use the setIdentifiers(true), to make netcurl add information about it self after your custom string. If you don't have any custom string the identifier enabling resets the default user agent data and sanitzes the user-agent output to the netcurl information only. A second argument, if true also adds the current active PHP-version to the user-agent. This is not enabled by default for secure reasons.

Identifiers are only available from NetWrapper as NetWrapper, or the former MODULE_CURL, is an auto selective helper.

WSDL Cache

netcurl 6.1 has a "more native" support for wsdl caching on fly and are configurable this way too. The default setup for the SoapClient - which is being set from WrapperConfig, is to not cache any soap-request. When testing things, it's really optimal to not cache request so that you always are guaranteed to fetch responses as you are expecting them. However, in productions you may want to consider the cache as there are time to save here. This is indirectly being made via the WrapperConfig like this for example:

```
$soapWrapper = (new SoapClientWrapper($this->wsdl))->setWsdlCache(WSDL_CACHE_DISK);
```

Normally, this could be done via WrapperConfig, but as we are in the SoapWrapper at this moment, I've decided to make some of the calls available directly from that wrapper. Mostly because this may be the only place you really need to configure a wsdl-cache. This is built on a magic, which means that methods that is locally created or resides in WrapperConfig, will be passed over to each local methods. The rest of the calls is considered soapcalls to a remote API. **setWsdlCache** also actually takes two arguments - the first arguments decides which kind of cache you wish to use, and the second argument is a ttl in seconds which sets up the time for the cache to live, before it gets renewed. You can see more about this here at https://www.php.net/manual/en/soap.configuration.php#ini.soap.wsdl-cache-ttl - just make sure that ini_set isn't blocked or those settings won't be pushed out anywhere.

Authorization in cached mode vs uncached mode

The issue **NETCURL-274** - Getting issue details... STATUS reveals that by caching wsdl requests, you may also move authorization failures around a bit in the API. For example, when soapcalls are uncached, both authorization and downloading of the wsdl document happens in the same time. By means, authorization is being made while the document is downloading. By submitting wrong credentials at this moment (depending on the API of course) the client will immediately receive a 401 Unauthorized. There is known bug from 2006 ⚠️ that is unfixed (beacuse it will be fixed in PHP >7.x somewhere) - see **NETCURL-67** - Getting issue details... STATUS or the bugreport here.

When you choose to cache wsdl requests, this error will be delayed until the real soap-method are actually executed. At this moment, there are still plenty of errors that may be an issue for systems that expects error codes to be thrown. When sending wrong credentials in "cached mode" a SoapFault will be thrown but with the code HTTP, instead of a real getCode(). It's still possible to catch this error, but it's much harder to figure out the content of it, unless you check out the faultstring (as told in the 38703). What would actually happen in multilingual systems? If the faultstring by any reason is translated or misspelled, you won't even be able to catch the message.

What netcurl does during this special occurance is to extract the http-heder from the SoapClient and tries to extract the top HTTP/x.x 401 Unauthorized response. If the code is actually 401, it will throw a proper Exception(Http_Head_String, Http_Head_Code) instead.

Multiple requests

Thanks to the new updated interface, the module can now request multiple urls in one call. Instead of entering a single string based url into NetWrapper, you can enter urls as an array. This is only supported by NetWrapper and CurlWrapper as of may 2020 (since curl is the only engine that supports multiple calls, it is only there the focus has been put onm this matter). When entering urls as an array, parsed data and bodies can be fetched by entering each url in the method. Compared to single requests ...

```
$wrapper = (new NetWrapper())->request('https://test.com');
$wrapper->getBody();
$wrapper->getParsed();
```

... multi requests almost looks the same, except that push in which URL you want to read the response from.

```
$wrapper = (new NetWrapper())->request(['https://test.com']);
$wrapper->getBody('https://test.com');
$wrapper->getParsed('https://test.com');
```

Different exception executions

Exceptions are also handled differently for multiple calls. The default behaviour is to throw all exceptions when all requests are fulfilled. This happens since the entire process should be protected and not fail, due to only one request.

When exceptions are thrown there are two ways of getting the failed errors.

If you wrapped the request in a online request like this ...

genericTest.php – multiCurlErrorHandlingMultiError (curl_multi)

```
try {
            (new NetWrapper())->request(
                [
                    'http://ipv4.netcurl.org/http.php?code=200&message=Funktionsduglig',
                    'http://ipv4.netcurl.org/http.php?code=500&message=Kass',
                    'http://ipv4.netcurl.org/http.php?code=500&message=Trasig',
                    'http://ipv4.netcurl.org/http.php?code=201&message=Mittemellan',
                ]
            );
        }
```

... you can still extract the entire exception block from the catch:

exception results

```
            // Requesting data from https://this.test.com
        } catch (ExceptionHandler $e) {
            // If one fails, responses can be extracted from here, but should normally be analyzed instead.
            /** @var ExceptionHandler $extendedException */
            $extendedException = $e->getExtendException();
            $properParsed = $extendedException->getParsed('http://this.test.com');
        }
```

By catching ExceptionHandler instead of a regular Exception, you can fetch the currently used wrapper by getExtendException(). In this case, that variable contains the CurlWrapper object, from which you can continue get parsed data. The way exceptions are handled here is the same for simple requests. The second way, if you need to keep the wrapper intact regardless of exceptions this works aswell:

genericTest.php – regularExceptionTest

```
        $wrapper = new NetWrapper();
        try {
            $wrapper->request('http://ipv4.netcurl.org/http.php?code=404&message=Error404+Generated');
        } catch (ExceptionHandler $e) {
            $code = $e->getCode();
        }
        $reqCode = $wrapper->getCode();
```