VCL (/vcl)

Cryptographic (/vcl/cryptographic/)

Notes

In base64 decoding, the output theoretically could be in binary but is interpreted as a string. So if the binary output contains '\0' then it could be truncated.

The time based One-Time Password algorithm initializes the HMAC using the key and appropriate hash type. Then it hashes the message

`(<time now in seconds since UNIX epoch> / <interval>) + <offset>`

as a 64bit unsigned integer (little endian) and base64 encodes the result.

Examples

One-Time Password Validation (Token Authentication)

Use this to validate tokens with a URL format like the following:

`http://cname-to-fastly/video.mp4?6h2YUl1CB4C50SbkZ0E6U3dZGjh+84dz3+Zope2Uhik=`

Example implementations for token generation in various languages can be [found in GitHub](https://github.com/fastly/token-functions) (https://github.com/fastly/token-functions).

Example VCL

```
 1   sub vcl_recv {
 2
 3     /* make sure there is a token */
 4     if (req.url !~ "[?&]token=([^&]+)") {
 5       error 403;
 6     }
 7
 8     if (re.group.1 != digest.time_hmac_sha256("RmFzdGx5IFRva2VuIFRlc3Q=", 60, 0) &&
 9         re.group.1 != digest.time_hmac_sha256("RmFzdGx5IFRva2VuIFRlc3Q=", 60, -1)) {
10       error 403;
11     }
12
13   #FASTLY recv
14
15     ...
16   }
```

Signature

```
1   set resp.http.x-data-sig = digest.hmac_sha256("secretkey",resp.http.x-data);
```

Base64 decoding

A snippet like this in `vcl_error` would set the response body to the value of the request header field named `x-parrot` after base64-decoding the value:

```
1   synthetic digest.base64_decode(req.http.x-parrot);
```

However, if the base64-decoded string contains a NUL byte (0x00), then that byte and any bytes following it will not be included in the response. Keep that in mind if you intend to send a synthetic response that contains binary data. There is currently no way to send a synthetic response containing a NUL byte.

Cryptographic Functions

📄 digest.awsv4_hmac() (/vcl/functions/digest-awsv4-hmac/)

Returns an AWSv4 message authentication code (https://docs.aws.amazon.com/AmazonS3/latest/API/sig-v4-authenticating-requests.html#signing-request-intro) based on the supplied `key` and `string`. This function automatically prepends "AWS4" in front of the secret access key (the first function parameter) as required by the protocol. This function does not support binary data for its `key` or `string` parameters.

Format

```
STRING (/vcl/types/string)
digest.awsv4_hmac(STRING key, STRING date_stamp, STRING region, STRING service, STRING string)
```

Examples

```
1   declare local var.signature STRING;
2   set var.signature = digest.awsv4_hmac(
3       "wJalrXUtnFEMI/K7MDENG+bPxRfiCYEXAMPLEKEY",
4       "20120215",
5       "us-east-1",
6       "iam",
7       "hello");
```

📄 digest.base64_decode() (/vcl/functions/digest-base64-decode/)

Decode Base64. Returns a string.

Format

```
STRING (/vcl/types/string)
digest.base64_decode(STRING input)
```

🗎 digest.base64() (/vcl/functions/digest-base64/)

Base64 (https://en.wikipedia.org/wiki/Base64) encoding. Returns the base64-encoded version of the input-string.

Format

```
STRING (/vcl/types/string)
digest.base64(STRING input)
```

🗎 digest.base64url_decode() (/vcl/functions/digest-base64url-decode/)

Decode Base64 with url safe characters in. Returns a string.

Format

```
STRING (/vcl/types/string)
digest.base64url_decode(STRING input)
```

🗎 digest.base64url_nopad_decode() (/vcl/functions/digest-base64url-nopad-decode/)

Decode Base64 with url safe characters. Returns a string. Identical to base64_url_decode.

Format

```
STRING (/vcl/types/string)
digest.base64url_nopad_decode(STRING input)
```

🗎 digest.base64url_nopad() (/vcl/functions/digest-base64url-nopad/)

Base64 (https://en.wikipedia.org/wiki/Base64#URL_applications) encoding. Returns the base64-encoded version of the input-string. Replaces +/ with -_ for url safety. Has no length padding (https://en.wikipedia.org/wiki/Base64#Padding).

Format

```
STRING (/vcl/types/string)
digest.base64url_nopad(STRING input)
```

📄 digest.base64url() (/vcl/functions/digest-base64url/)

Base64 (https://en.wikipedia.org/wiki/Base64#URL_applications) encoding. Returns the base64-encoded version of the input-string. Replaces +/ with -_ for url safety.

Format

```
STRING (/vcl/types/string)
digest.base64url(STRING input)
```

📄 digest.hash_crc32() (/vcl/functions/digest-hash-crc32/)

Use a 32 bit Cyclic Redundancy Checksum (https://en.wikipedia.org/wiki/CRC32). Returns a hex-encoded string.

Format

```
STRING (/vcl/types/string)
digest.hash_crc32(STRING input)
```

📄 digest.hash_crc32b() (/vcl/functions/digest-hash-crc32b/)

A reversed CRC32 (for compatibility with some PHP applications (http://php.net/manual/en/function.hash-file.php#104836)). Returns a hex-encoded string.

Format

```
STRING (/vcl/types/string)
digest.hash_crc32b(STRING input)
```

📄 digest.hash_md5() (/vcl/functions/digest-hash-md5/)

Use the MD5 (https://en.wikipedia.org/wiki/MD5) hash. Returns a hex-encoded string.

Format

```
STRING (/vcl/types/string)
digest.hash_md5(STRING input)
```

Examples

```
1  declare local var.hash_md5 STRING;
2  set var.hash_md5 = digest.hash_md5("123456789");
3  # var.hash_md5 now contains "25f9e794323b453885f5181f1b624d0b"
```

📄 digest.hash_sha1() (/vcl/functions/digest-hash-sha1/)

Use the SHA1 (https://en.wikipedia.org/wiki/Secure_Hash_Algorithm) hash. Returns a hex-encoded string.

Format

```
STRING (/vcl/types/string)
digest.hash_sha1(STRING input)
```

Examples

```
1  declare local var.hash_sha1 STRING;
2  set var.hash_sha1 = digest.hash_sha1("123456789");
3  # var.hash_sha1 now contains "f7c3bc1d808e04732adf679965ccc34ca7ae3441"
```

📄 digest.hash_sha224() (/vcl/functions/digest-hash-sha224/)

Use the SHA224 (https://en.wikipedia.org/wiki/Secure_Hash_Algorithm) hash. Returns a hex-encoded string.

Format

```
STRING (/vcl/types/string)
digest.hash_sha224(STRING input)
```

Examples

```
1  declare local var.hash_sha224 STRING;
2  set var.hash_sha224 = digest.hash_sha224("123456789");
3  # var.hash_sha224 now contains "9b3e61bf29f17c75572fae2e86e17809a4513d07c8a18152acf34
   521"
```

📄 digest.hash_sha256() (/vcl/functions/digest-hash-sha256/)

Use the SHA256 (https://en.wikipedia.org/wiki/Secure_Hash_Algorithm) hash. Returns a hex-encoded string.

Format

```
STRING (/vcl/types/string)
digest.hash_sha256(STRING input)
```

Examples

```
1  declare local var.hash_sha256 STRING;
2  set var.hash_sha256 = digest.hash_sha256("123456789");
3  # var.hash_sha256 now contains "15e2b0d3c33891ebb0f1ef609ec419420c20e320ce94c65fbc8c3
   312448eb225"
```

📄 digest.hash_sha384() (/vcl/functions/digest-hash-sha384/)

Use the SHA384 (https://en.wikipedia.org/wiki/Secure_Hash_Algorithm) hash. Returns a hex-encoded string.

Format

```
STRING (/vcl/types/string)
digest.hash_sha384(STRING input)
```

Examples

```
1   declare local var.hash_sha384 STRING;
2   set var.hash_sha384 = digest.hash_sha384("123456789");
3   # var.hash_sha384 now contains "eb455d56d2c1a69de64e832011f3393d45f3fa31d6842f21af92d
    2fe469c499da5e3179847334a18479c8d1dedea1be3"
```

📄 digest.hash_sha512() (/vcl/functions/digest-hash-sha512/)

Use the SHA512 (https://en.wikipedia.org/wiki/Secure_Hash_Algorithm) hash. Returns a hex-encoded string.

Format

```
STRING (/vcl/types/string)
digest.hash_sha512(STRING input)
```

Examples

```
1   declare local var.hash_sha512 STRING;
2   set var.hash_sha512 = digest.hash_sha512("123456789");
3   # var.hash_sha512 now contains "d9e6762dd1c8eaf6d61b3c6192fc408d4d6d5f1176d0c29169bc2
    4e71c3f274ad27fcd5811b313d681f7e55ec02d73d499c95455b6b5bb503acf574fba8ffe85"
```

📄 digest.hmac_md5_base64() (/vcl/functions/digest-hmac-md5-base64/)

Hash-based message authentication code (https://en.wikipedia.org/wiki/Hash-based_message_authentication_code) using MD5. Returns a base64-encoded (https://en.wikipedia.org/wiki/Base64) string.

Format

```
STRING (/vcl/types/string)
digest.hmac_md5_base64(STRING key, STRING input)
```

Examples

```
1   declare local var.hmac_md5_base64 STRING;
2   set var.hmac_md5_base64 = digest.hmac_md5_base64("key", "input");
3   # var.hmac_md5_base64 now contains "cZ/HW66QBNnoQqSxW4KMBg=="
```

📄 digest.hmac_md5() (/vcl/functions/digest-hmac-md5/)

Hash-based message authentication code (https://en.wikipedia.org/wiki/Hash-based_message_authentication_code) using MD5. Returns a hex-encoded string prepended with 0x.

Format

```
STRING (/vcl/types/string)
digest.hmac_md5(STRING key, STRING input)
```

Examples

```
1   declare local var.hmac_md5 STRING;
2   set var.hmac_md5 = digest.hmac_md5("key", "input");
3   # var.hmac_md5 now contains "0x719fc75bae9004d9e842a4b15b828c06"
```

📄 digest.hmac_sha1_base64() (/vcl/functions/digest-hmac-sha1-base64/)

Hash-based message authentication code (https://en.wikipedia.org/wiki/Hash-based_message_authentication_code) using SHA-1 (https://en.wikipedia.org/wiki/Secure_Hash_Algorithm). Returns a base64-encoded (https://en.wikipedia.org/wiki/Base64) string.

Format

```
STRING (/vcl/types/string)
digest.hmac_sha1_base64(STRING key, STRING input)
```

Examples

```
1   declare local var.hmac_sha1_base64 STRING;
2   set var.hmac_sha1_base64 = digest.hmac_sha1_base64("key", "input");
3   # var.hmac_sha1_base64 now contains "hRO7NVB2zOKuXrnzmatcr9unyKI="
```

📄 digest.hmac_sha1() (/vcl/functions/digest-hmac-sha1/)

Hash-based message authentication code (https://en.wikipedia.org/wiki/Hash-based_message_authentication_code) using SHA-1 (https://en.wikipedia.org/wiki/Secure_Hash_Algorithm). Returns a hex-encoded string prepended with 0x.

Format

```
STRING (/vcl/types/string)
digest.hmac_sha1(STRING key, STRING input)
```

Examples

```
1  declare local var.hmac_sha1 STRING;
2  set var.hmac_sha1 = digest.hmac_sha1("key", "input");
3  # var.hmac_sha1 now contains "0x8513bb355076cce2ae5eb9f399ab5cafdba7c8a2"
```

📄 digest.hmac_sha256_base64() (/vcl/functions/digest-hmac-sha256-base64/)

Hash-based message authentication code (https://en.wikipedia.org/wiki/Hash-based_message_authentication_code) using SHA-256 (https://en.wikipedia.org/wiki/Secure_Hash_Algorithm). Returns a base64-encoded (https://en.wikipedia.org/wiki/Base64) string.

Format

```
STRING (/vcl/types/string)
digest.hmac_sha256_base64(STRING key, STRING input)
```

Examples

```
1  declare local var.hmac_sha256_base64 STRING;
2  set var.hmac_sha256_base64 = digest.hmac_sha256_base64("key", "input");
3  # var.hmac_sha256_base64 now contains "ngiewTr4gaisInpzbD58SQ6jtK/KDF+D3/Y5O2g6cuM="
```

📄 digest.hmac_sha256() (/vcl/functions/digest-hmac-sha256/)

Hash-based message authentication code (https://en.wikipedia.org/wiki/Hash-based_message_authentication_code) using SHA-256 (https://en.wikipedia.org/wiki/Secure_Hash_Algorithm). Returns a hex-encoded string prepended with 0x.

Format

```
STRING (/vcl/types/string)
digest.hmac_sha256(STRING key, STRING input)
```

Examples

```
1  declare local var.hmac_sha256 STRING;
2  set var.hmac_sha256 = digest.hmac_sha256("key", "input");
3  # var.hmac_sha256 now contains "0x9e089ec13af881a8ac227a736c3e7c490ea3b4afca0c5f83dff
   6393b683a72e3"
```

📄 digest.rsa_verify() (/vcl/functions/digest-rsa-verify/)

A boolean function that returns true if the RSA signature of `payload` using `public_key` matches `digest`. The `hash_method` parameter selects the digest function to use. It can be `sha256`, `sha384`, `sha512`, or `default` (`default` is equivalent to `sha256`). The `STRING_LIST` parameter in the payload/digest could reference headers such as `req.http.payload` and `req.http.digest`. The `base64_method` parameter is optional. It can be `standard`, `url`, `url_nopad`, or `default` (`default` is equivalent to `url_nopad`).

Format

```
BOOL (/vcl/types/bool)
digest.rsa_verify(ID hash_method, STRING_LIST public_key, STRING_LIST payload, STRING_LIST dige
st [, ID base64_method ])
```

Examples

```
1  if (digest.rsa_verify(sha256, {"-----BEGIN PUBLIC KEY-----
2  aabbccddIieEffggHHhEXAMPLEPUBLICKEY
3  -----END PUBLIC KEY-----"}, req.http.payload, req.http.digest, url_nopad)) {
4    set req.http.verified = "Verified";
5  } else {
6    set req.http.verified = "Not Verified";
7  }
8  error 900;
```

📄 digest.secure_is_equal() (/vcl/functions/digest-secure-is-equal/)

A boolean function that returns true if s1 and s2 are equal. The comparison is done in constant time to defend against timing attacks.

Format

```
BOOL (/vcl/types/bool)
digest.secure_is_equal(STRING_LIST s1, STRING_LIST s2)
```

📄 digest.time_hmac_md5() (/vcl/functions/digest-time-hmac-md5/)

Time based One Time Password using MD5. Returns base64 encoded output.

Format

```
STRING (/vcl/types/string)
digest.time_hmac_md5(STRING key, INTEGER interval, INTEGER offset)
```

🗎 digest.time_hmac_sha1() (/vcl/functions/digest-time-hmac-sha1/)

Time based One Time Password using SHA1. Returns base64 encoded output.

Format

```
STRING (/vcl/types/string)
digest.time_hmac_sha1(STRING key, INTEGER interval, INTEGER offset)
```

🗎 digest.time_hmac_sha256() (/vcl/functions/digest-time-hmac-sha256/)

Time based One Time Password using SHA256. Returns base64 encoded output.

Format

```
STRING (/vcl/types/string)
digest.time_hmac_sha256(STRING key, INTEGER interval, INTEGER offset)
```

Date and time (/vcl/date-and-time/)

Date and time Functions

🗎 std.integer2time() (/vcl/functions/std-integer2time/)

Converts an integer, representing seconds since the UNIX Epoch (https://en.wikipedia.org/wiki/Unix_time), to a time variable.

If the time argument is invalid then this returns a time value which stringifies to: `datetime out of bounds`.

To convert a string, use `std.time()` (/vcl/functions/std-time/) instead.

Format

```
TIME (/vcl/types/time)
std.integer2time(INTEGER time)
```

Examples

```
1  # var.once will represent: Mon, 02 Jan 2006 22:04:05 GMT
2  declare local var.once TIME;
3  set var.once = std.integer2time(1136239445);
```

📄 std.time() (/vcl/functions/std-time/)

Converts a string to a time variable.

The following string formats are supported:

- `Mon, 02 Jan 2006 22:04:05 GMT`, RFC 822 (https://tools.ietf.org/html/rfc822) and RFC 1123 (https://tools.ietf.org/html/rfc1123)

- `Monday, 02-Jan-06 22:04:05 GMT`, RFC 850 (https://tools.ietf.org/html/rfc850)

- `Mon Jan  2 22:04:05 2006`, ANSI-C asctime() (https://www.unix.com/man-page/FreeBSD/3/asctime/)

- `2006-01-02 22:04:05`, an ISO 8601 (https://en.wikipedia.org/wiki/ISO_8601) subset

- `1136239445.00`, seconds since the UNIX Epoch (https://en.wikipedia.org/wiki/Unix_time)

- `1136239445`, seconds since the UNIX Epoch (https://en.wikipedia.org/wiki/Unix_time)

The only time zone supported is `GMT`.

If the string does not match one of those formats, then the fallback variable is returned instead.

Format

```
TIME (/vcl/types/time)
std.time(STRING s, TIME fallback)
```

Examples

```
1  # var.string will represent: Mon, 02 Jan 2006 22:04:05 GMT
2  declare local var.string TIME;
3  set var.string = std.time("Mon, 02 Jan 2006 22:04:05 GMT", std.integer2time(-1));
```

```
1  # var.integer will represent: Mon, 02 Jan 2006 22:04:05 GMT
2  declare local var.integer TIME;
3  set var.integer = std.time("1136239445", std.integer2time(-1));
```

```
1  # var.invalid will represent: datetime out of bounds
2  declare local var.invalid TIME;
3  set var.invalid = std.time("Not a date", std.integer2time(-1));
```

📄 strftime() (/vcl/functions/strftime/)

Formats a time to a string. This uses standard POSIX strftime formats (https://www.unix.com/man-page/FreeBSD/3/strftime/).

Format

```
STRING (/vcl/types/string)
strftime(STRING format, TIME time)
```

Examples

```
1  # Concise format, e.g.: 2006-01-02 22:04
2  set resp.http.now = strftime({"%Y-%m-%d %H:%M"}, now);
```

```
1  # RFC 5322 format, e.g.: Mon, 02 Jan 2006 22:04:05 +0000
2  set resp.http.start = strftime({"%a, %d %b %Y %T %z"}, time.start);
```

```
1  # ISO 8601 format, e.g.: 2006-01-02T22:04:05Z
2  set resp.http.end = strftime({"%Y-%m-%dT%H:%M:%SZ"}, time.end);
```

📄 time.add() (/vcl/functions/time-add/)

Adds a relative time to a time.

Format

```
TIME (/vcl/types/time)
time.add(TIME t1, TIME t2)
```

Examples

```
1  # var.one_day_later will represent the same time tomorrow
2  declare local var.one_day_later TIME;
3  set var.one_day_later = time.add(now, 1d);
```

📄 time.hex_to_time() (/vcl/functions/time-hex-to-time/)

This specialized function takes a hexadecimal string value, divides by `divisor` and interprets the result as seconds since the UNIX Epoch (https://en.wikipedia.org/wiki/Unix_time).

Format

```
TIME (/vcl/types/time)
time.hex_to_time(INTEGER divisor, STRING dividend)
```

Examples

```
1  # var.hextime will represent: Mon, 02 Jan 2006 22:04:05 GMT
2  declare local var.hextime TIME;
3  set var.hextime = time.hex_to_time(1, "43b9a355");
```

📄 time.is_after() (/vcl/functions/time-is-after/)

Returns true if `t1` is after `t2`. (Normal timeflow and causality required.)

Format

```
BOOL (/vcl/types/bool)
time.is_after(TIME t1, TIME t2)
```

Examples

```
1  if (time.is_after(time.add(now, 10m), now)) {
2    ...
3  }
```

📄 time.sub() (/vcl/functions/time-sub/)

Subtracts a relative time from a time.

Format

```
TIME (/vcl/types/time)
time.sub(TIME t1, TIME t2)
```

Examples

```
1  # var.one_day_earlier will represent the same time yesterday
2  declare local var.one_day_earlier TIME;
3  set var.one_day_earlier = time.sub(now, 1d);
```

Date and time Variables

📄 now.sec (/vcl/variables/now-sec/)

Like the `now` variable, but in seconds since the UNIX Epoch
(https://en.wikipedia.org/wiki/Unix_time).

Readable From

All subroutines

📄 now (/vcl/variables/now/)

The current time in RFC 1123 format (https://www.ietf.org/rfc/rfc1123.txt) format.

Readable From

All subroutines

📄 time.elapsed.msec_frac (/vcl/variables/time-elapsed-msec-frac/)

The time the request started in milliseconds since the last whole second.

Readable From

- `vcl_deliver`
- `vcl_log`

📄 time.elapsed.msec (/vcl/variables/time-elapsed-msec/)

The time since the request start in milliseconds.

Readable From

- `vcl_deliver`
- `vcl_log`

📄 time.elapsed.sec (/vcl/variables/time-elapsed-sec/)

The time since the request start in seconds.

Readable From

- `vcl_deliver`
- `vcl_log`

📄 time.elapsed.usec_frac (/vcl/variables/time-elapsed-usec-frac/)

The time the request started in microseconds since the last whole second.

Readable From

- `vcl_deliver`
- `vcl_log`

📄 time.elapsed.usec (/vcl/variables/time-elapsed-usec/)

The time since the request start in microseconds.

Readable From

- `vcl_deliver`

- `vcl_log`

📄 time.elapsed (/vcl/variables/time-elapsed/)

The time since the request start, using RFC 1123 format (https://www.ietf.org/rfc/rfc1123.txt). Also useful for strftime.

Readable From

- `vcl_deliver`

- `vcl_log`

📄 time.end.msec_frac (/vcl/variables/time-end-msec-frac/)

The time the request started in milliseconds since the last whole second.

Readable From

- `vcl_deliver`

- `vcl_log`

📄 time.end.msec (/vcl/variables/time-end-msec/)

The time the request ended in milliseconds since the UNIX Epoch (https://en.wikipedia.org/wiki/Unix_time).

Readable From

- `vcl_deliver`

- `vcl_log`

📄 time.end.sec (/vcl/variables/time-end-sec/)

The time the request ended in seconds since the UNIX Epoch (https://en.wikipedia.org/wiki/Unix_time).

Readable From

- `vcl_deliver`

- `vcl_log`

📄 time.end.usec_frac (/vcl/variables/time-end-usec-frac/)

The time the request started in microseconds since the last whole second.

Readable From

- `vcl_deliver`

- `vcl_log`

📄 time.end.usec (/vcl/variables/time-end-usec/)

The time the request ended in microseconds since the UNIX Epoch (https://en.wikipedia.org/wiki/Unix_time).

Readable From

- `vcl_deliver`

- `vcl_log`

📄 time.end (/vcl/variables/time-end/)

The time the request ended, using RFC 1123 format (https://www.ietf.org/rfc/rfc1123.txt). Also useful for strftime.

Readable From

- `vcl_deliver`

- `vcl_log`

📄 time.start.msec_frac (/vcl/variables/time-start-msec-frac/)

The time the request started in milliseconds since the last whole second, after TLS termination.

Readable From

All subroutines

📄 time.start.msec (/vcl/variables/time-start-msec/)

The time the request started in milliseconds since the UNIX Epoch (https://en.wikipedia.org/wiki/Unix_time), after TLS termination.

Readable From

All subroutines

📄 time.start.sec (/vcl/variables/time-start-sec/)

The time the request started in seconds since the UNIX Epoch (https://en.wikipedia.org/wiki/Unix_time), after TLS termination.

Readable From

All subroutines

📄 time.start.usec_frac (/vcl/variables/time-start-usec-frac/)

The time the request started in microseconds since the last whole second, after TLS termination.

Readable From

All subroutines

📄 time.start.usec (/vcl/variables/time-start-usec/)

The time the request started in microseconds since the UNIX Epoch (https://en.wikipedia.org/wiki/Unix_time), after TLS termination.

Readable From

All subroutines

📄 time.start (/vcl/variables/time-start/)

The time the request started, after TLS termination, using RFC 1123 format (https://www.ietf.org/rfc/rfc1123.txt).

Readable From

All subroutines

📄 time.to_first_byte (/vcl/variables/time-to-first-byte/)

The time interval since the request started up to the point before the `vcl_deliver` function ran. When used in a string context, an RTIME variable like this one will be formatted as a number in seconds with 3 decimal digits of precision. In `vcl_deliver` this interval will be very close to `time.elapsed`. In `vcl_log`, the difference between `time.elapsed` and `time.to_first_byte` will be the time that it took to send the response body.

Readable From

- `vcl_deliver`

- `vcl_log`

Edge Side Includes (ESI) (/vcl/esi/)

> ### Edge Side Includes (ESI) Variables

📄 req.esi (/vcl/variables/req-esi/)

Whether or not to enable ESI processing during this request. Using `set req.esi = true;` will enable ESI processing.

Type

`BOOL (/vcl/types/bool)`

Readable From

- `vcl_recv`

- `vcl_fetch`

- `vcl_deliver`

- `vcl_error`

📄 req.topurl (/vcl/variables/req-topurl/)

In an ESI subrequest, returns the URL of the top-level request.

Type

`STRING (/vcl/types/string)`

Accessibility

Readable From

All subroutines

Geolocation (/vcl/geolocation/)

> ℹ **NOTE:** While Fastly exposes these geographic variables, we cannot guarantee their accuracy. The variables are based on available geographic data and are intended to provide an approximate location of where requests might be coming from, rather than an exact location. The postal code associated with an IP address is the most granular level of geographic data available.

> ℹ **NOTE:** Geolocation information, including data streamed by our log streaming service (/guides/streaming-logs/), is intended to be used only in connection with your use of Fastly services. Use of geolocation data for other purposes may require the permission of a IP geolocation dataset vendor, such as Digital Element (https://www.digitalelement.com/end-user-license-agreement-eula/).

> ★ **TIP:** If you're updating your configurations from older version of the geolocation variables, be sure to read our migration guide (/guides/migrations/migrating-geolocation-variables-to-the-new-dataset).

Using geographic variables with shielding

If you have shielding (/guides/performance-tuning/shielding) enabled, you should set the following variable before using geographic variables:

```
1   set client.geo.ip_override = req.http.Fastly-Client-IP;
```

Geolocation Variables

📄 client.as.name (/vcl/variables/client-as-name/)

The name of the organization associated with `client.as.number`.

Readable From

All subroutines

📄 client.as.number (/vcl/variables/client-as-number/)

The autonomous system (AS) (https://en.wikipedia.org/wiki/Autonomous_system_(Internet)) number associated with this IP address.

Readable From

All subroutines

📄 client.geo.area_code (/vcl/variables/client-geo-area-code/)

The telephone area code associated with the IP address. These are only available for IP addresses in the United States.

Readable From

All subroutines

📄 client.geo.city.ascii (/vcl/variables/client-geo-city-ascii/)

An alias of `client.geo.city`.

Readable From

All subroutines

📄 client.geo.city.utf8 (/vcl/variables/client-geo-city-utf8/)

The city or town name associated with the IP address, encoded using the UTF-8 character encoding.

Readable From

All subroutines

📄 client.geo.city (/vcl/variables/client-geo-city/)

The city or town name associated with the IP address, encoded using the ASCII character encoding (a lowercase ASCII approximation of the original string with diacritics removed).

Readable From

All subroutines

📄 client.geo.conn_speed (/vcl/variables/client-geo-conn-speed/)

The type of connection speed (https://www.webopedia.com/quick_ref/internet_connection_types.asp) associated with the IP address. Possible values are: **broadband**, **cable**, **dialup**, **mobile**, **oc12**, **oc3**, **t1**, **t3**, **satellite**, **wireless**, **xdsl**.

Readable From

All subroutines

📄 client.geo.continent_code (/vcl/variables/client-geo-continent-code/)

A two-character code representing the continent associated with the IP address. Possible codes are: **AF** - Africa, **AS** - Asia, **EU** - Europe, **NA** - North America, **OC** - Oceania, **SA** - South America, **AN** - Antarctica.

Readable From

All subroutines

📄 client.geo.country_code (/vcl/variables/client-geo-country-code/)

A two-character ISO 3166-1 (https://en.wikipedia.org/wiki/ISO_3166-1) country code for the country associated with the IP address. The **US** country code is returned for IP addresses associated with overseas United States military bases.

📄 client.geo.country_code3 (/vcl/variables/client-geo-country-code3/)

A three-character ISO 3166-1 alpha-3 (https://en.wikipedia.org/wiki/ISO_3166-1_alpha-3) country code for the country associated with the IP address. The **USA** country code is returned for IP addresses associated with overseas United States military bases.

Readable From

All subroutines

📄 client.geo.country_name.ascii (/vcl/variables/client-geo-country-name-ascii/)

An alias of `client.geo.country_name`.

Readable From

All subroutines

📄 client.geo.country_name.utf8 (/vcl/variables/client-geo-country-name-utf8/)

The country name associated with the IP address, encoded using the UTF-8 character encoding.

Readable From

All subroutines

📄 client.geo.country_name (/vcl/variables/client-geo-country-name/)

The country name associated with the IP address, encoded using the ASCII character encoding (a lowercase ASCII approximation of the original string with diacritics removed).

Readable From

All subroutines

📄 client.geo.gmt_offset (/vcl/variables/client-geo-gmt-offset/)

The time zone offset from coordinated universal time (UTC) for the `client.geo.city` associated with the IP address.

Readable From

All subroutines

📄 client.geo.latitude (/vcl/variables/client-geo-latitude/)

The latitude associated with the IP address.

Readable From

All subroutines

📄 client.geo.longitude (/vcl/variables/client-geo-longitude/)

The longitude associated with the IP address.

Readable From

All subroutines

📄 client.geo.metro_code (/vcl/variables/client-geo-metro-code/)

The metro code associated with the IP address. Metro codes represent designated market areas (DMAs) in the United States and Germany, Independent Television Service (ITV) regions in the UK, department codes in France, South Korean administrative divisions (si, gun, gu or cities, counties, and districts), Chinese administrative regions (diji shi, or "region-level" cities), Russian federal districts, Norwegian municipalities, urban areas in New Zealand, and the Greater Capital City Statistical Area (GCCSA) and Significant Urban Areas (SUAs) in Australia.

Readable From

All subroutines

📄 client.geo.postal_code (/vcl/variables/client-geo-postal-code/)

The postal code associated with the IP address. These are available for some IP addresses in Australia, Canada, France, Germany, Italy, Spain, Switzerland, the United Kingdom, and the United States. We return the first 3 characters for Canadian postal codes. We return the first 2-4 characters (outward code) for postal codes in the United Kingdom.

Readable From

All subroutines

📄 client.geo.region (/vcl/variables/client-geo-region/)

The ISO 3166-2 (https://en.wikipedia.org/wiki/ISO_3166-2) region code associated with the IP address.

Readable From

All subroutines

📄 server.datacenter (/vcl/variables/server-datacenter/)

A code representing one of <u>Fastly's POP locations (/guides/basic-concepts/fastly-pop-locations)</u>.

Readable From

All subroutines

📄 server.region (/vcl/variables/server-region/)

A code representing the general region of the world in which the POP location resides. One of the following:

Region Name	Approximate Geographic Location of Fastly POPs
APAC	Australia and New Zealand
Asia	throughout the Asian continent (except India)
Asia-South	southern Asia
EU-Central	the central European continent
EU-East	the eastern European continent
EU-West	the western European continent
North-America	Canada
SA-East	eastern South America
SA-South	southern South America
South-Africa	the southern regions of Africa
US-Central	the central United States
US-East	the eastern United States
US-West	the western United States

Readable From

All subroutines

Miscellaneous (/vcl/miscellaneous/)

Miscellaneous features

Feature	Description
`goto`	Performs a one-way transfer of control to another line of code. See the <u>example</u> for more information.

Feature	Description
`return`	Returns (with no return value) from a custom subroutine to exit early. See the example for more information.

Examples

Use the following examples to learn how to implement the features.

Goto

Similar to some programming languages, `goto` statements in VCL allow you perform a one-way transfer of control to another line of code. When using `goto`, jumps must always be forward, rather than to an earlier part of code.

This act of "jumping" allows you to do things like perform logical operations or set headers before returning lookup, error, or pass actions. These statements also make it easier to do things like jump to common error handling blocks before returning from a function. The `goto` statement works in custom subroutines.

```
1  sub vcl_recv {
2    if (!req.http.Foo) {
3      goto foo;
4    }
5
6  foo:
7    set req.http.Foo = "1";
8  }
```

Return

You can use `return` to exit early from a custom subroutine.

```
1  sub custom_subroutine {
2    if (req.http.Cookie:user_id) {
3      return;
4    }
5
6    # do a bunch of other stuff
7  }
```

Miscellaneous Functions

📄 cstr_escape() (/vcl/functions/cstr-escape/)

Escapes bytes unsafe for printing from a string using C-style escape sequences.

> ★ **TIP:** If you are escaping JSON strings, use `json.escape()` (/vcl/functions/json-escape/)
> instead.

Format

```
STRING (/vcl/types/string)
cstr_escape(STRING string)
```

Examples

```
1  # var.escaped is set to: city="london"
2  declare local var.escaped STRING;
3  set var.escaped = "city=%22" + cstr_escape(client.geo.city.ascii) + "%22";
```

📄 http_status_matches() (/vcl/functions/http-status-matches/)

Determines whether or not an HTTP status code matches a pattern. The arguments are an integer
(usually `beresp.status` or `resp.status`) and a comma-separated list of status codes, optionally
prefixed by a `!` to negate the match. It returns `true` or `false`.

Format

```
BOOL (/vcl/types/bool)
http_status_matches(INTEGER status, STRING fmt)
```

Examples

```
1  if (http_status_matches(beresp.status, "!200,304")) {
2    restart;
3  }
```

📄 if() (/vcl/functions/if/)

Implements a ternary operator for strings; if the expression is true, it returns `value-when-true`; if
the expression is false, it returns `value-when-false`. When the `if(x, value-when-true, value-
when-false);` argument is true, the `value-when-true` is returned. Otherwise, the `value-when-
false` is returned.

You can use `if()` as a construct to make simple conditional expressions more concise.

Format

```
STRING (/vcl/types/string)
if(BOOL expression, STRING value-when-true, STRING value-when-false)
```

Examples

```
1  set req.http.foo-status = if(req.http.foo, "present", "absent");
```

📄 json.escape() (/vcl/functions/json-escape/)

Escapes characters of a UTF-8 encoded Unicode string using JSON-style escape sequences.

Format

```
STRING (/vcl/types/string)
json.escape(STRING string)
```

Examples

```
1  # var.json is set to: {"city": "london"}
2  declare local var.json STRING;
3  set var.json = "{%22city%22: %22" + json.escape(client.geo.city.utf8) + "%22}";
```

📄 prefixof() (/vcl/functions/prefixof/)

True if the string `haystack` begins with the string `needle`. An empty string is not considered a prefix.

Returns false otherwise.

Format

```
BOOL (/vcl/types/bool)
prefixof(STRING haystack, STRING needle)
```

Examples

```
1  set req.http.X-ps = std.prefixof("greenhouse", "green");
```

📄 regsub() (/vcl/functions/regsub/)

Replaces the first occurrence of `pattern`, which may be a Perl-compatible regular expression, in `input` with `replacement`. If no match is found, no replacement is made.

This function may fail to make a replacement if the regular expression recurses too heavily. Such a situation may occur with lookahead and lookbehind assertions, or other recursing non-regular expressions. In this case, `fastly.error` is set to "EREGRECUR".

If `pattern` contains grouping characters `(` or `)`, `re.group.<i>` produces the ith group. `re.group.0` always evaluates to the text matched by regsub itself.

This function is not prefixed with the `std.` namespace.

Format

```
STRING (/vcl/types/string)
regsub(STRING input, STRING pattern, STRING replacement)
```

Examples

```
1  set req.url = regsub(req.url, "?.*");
```

📄 regsuball() (/vcl/functions/regsuball/)

Replaces all occurrences of `pattern`, which may be a Perl-compatible regular expression, in `input` with `replacement`. If no matches are found, no replacements are made.

Once a replacement is made, substitutions continue from the end of the replaced buffer. Therefore, `regsuball("aa", "a", "aa")` will return a string "aaaa" instead of recursing indefinitely.

This function may fail to make a replacement if the regular expression recurses too heavily. Such a situation may occur with lookahead and lookbehind assertions, or other recursing non-regular expressions. In this case, `fastly.error` is set to "EREGRECUR".

This function is not prefixed with the `std.` namespace.

Format

```
STRING (/vcl/types/string)
regsuball(STRING input, STRING pattern, STRING replacement)
```

Examples

```
1  set req.url = regsuball(req.url, "\+", "%2520");
```

📄 setcookie.get_value_by_name() (/vcl/functions/setcookie-get-value-by-name/)

Returns a value associated with the `cookie_name` in the `Set-Cookie` header contained in the HTTP response indicated by `where`. An unset value is returned if cookie is not found or on error. In the `vcl_fetch` method, the `beresp` response is available. In `vcl_deliver` and `vcl_log`, the `resp` response is available.

If multiple cookies of the same name are present in the response, the value of the last one will be returned.

When this function does not have enough memory to succeed, the request is failed.

This function conforms to RFC6265 (http://tools.ietf.org/html/rfc6265#section-4.1.1).

Format

```
STRING (/vcl/types/string)
setcookie.get_value_by_name(ID where, STRING cookie_name)
```

Examples

```
1  set resp.http.MyValue = setcookie.get_value_by_name(resp, "myvalue");
```

📄 std.atoi() (/vcl/functions/std-atoi/)

Takes a string (which represents an integer) as an argument and returns its value.

Format

```
INTEGER (/vcl/types/integer)
std.atoi(STRING s)
```

Examples

```
1  if (std.atoi(req.http.X-Decimal) == 42) {
2    set req.http.X-TheAnswer = "Found";
3  }
```

📄 std.ip() (/vcl/functions/std-ip/)

An alias of `std.str2ip`.

Format

```
IP (/vcl/types/ip)
std.ip(STRING addr, STRING fallback)
```

📄 std.ip2str() (/vcl/functions/std-ip2str/)

Converts the IP address (v4 or v6) to a string.

Format

```
STRING (/vcl/types/string)
std.ip2str(IP ip)
```

Examples

```
1  if (std.ip2str(std.str2ip("192.0.2.1", "192.0.2.2")) == "192.0.2.1") {
```

📄 std.str2ip() (/vcl/functions/std-str2ip/)

Converts the string address to an IP address (v4 or v6). For example, `if (std.str2ip("192.0.2.1", "192.0.2.2") ~ my_acl) {` where `192.0.2.2` is the fallback. If conversion fails, the fallback will be returned. Note that only the first result from DNS resolution is returned.

Format

```
IP (/vcl/types/ip)
std.str2ip(STRING addr, STRING fallback)
```

📄 std.strlen() (/vcl/functions/std-strlen/)

Returns the length of the string. For example, `std.strlen("Hello world!");` will return `12` (because the string includes whitespaces and punctuation).

Format

```
INTEGER (/vcl/types/integer)
std.strlen(STRING s)
```

Examples

```
1  if (std.strlen(req.http.Cookie) > 1024) {
2    unset req.http.Cookie;
3  }
```

📄 std.strstr() (/vcl/functions/std-strstr/)

Finds the first occurrence of a byte string and returns its value.

Format

```
STRING (/vcl/types/string)
std.strstr(STRING haystack, STRING needle)
```

Examples

```
1  set req.http.X-qs = std.strstr(req.url, "?");
```

📄 std.strtol() (/vcl/functions/std-strtol/)

Converts a string to an integer, using the second argument as base. Base can be `2` to `36`, or `0`. A `0` base means that base 10 (decimal) is used, unless the string has a `0x` or `0` prefix, in which case base 16 (hexadecimal) and base 8 (octal) are used respectively. For example, `std.strtol("0xa0", 0)` will return `160`.

Format

```
INTEGER (/vcl/types/integer)
std.strtol(STRING s, INTEGER base)
```

Examples

```
1  if (std.strtol(req.http.X-HexValue, 16) == 42) {
2    set req.http.X-TheAnswer = "Found";
3  }
```

📄 std.tolower() (/vcl/functions/std-tolower/)

Changes the case of a string to lower case. For example, `std.tolower("HELLO");` will return `"hello"`.

Format

```
STRING (/vcl/types/string)
std.tolower(STRING_LIST s)
```

Examples

```
1  set beresp.http.x-nice = std.tolower("VerY");
```

📄 std.toupper() (/vcl/functions/std-toupper/)

Changes the case of a string to upper case. For example, `std.toupper("hello");` will return `"HELLO"`.

Format

```
STRING (/vcl/types/string)
std.toupper(STRING_LIST s)
```

Examples

```
1  set beresp.http.x-scream = std.toupper("yes!");
```

📄 subfield() (/vcl/functions/subfield/)

Provides a means to access subfields from a header like `Cache-Control`, `Cookie`, and `Edge-Control` or individual parameters from the query string.

The optional separator character parameter defaults to `,`. It can be any one-character constant. For example, `;` is a useful separator for extracting parameters from a Set-Cookie field.

This functionality is also achievable by using the `:` accessor within a variable name. When the subfield is a valueless token (like "private" in the case of `Cache-Control: max-age=1200, private`), an empty string is returned. The `:` accessor also works for retrieving variables in a cookie.

This function is not prefixed with the `std.` namespace.

Format

```
STRING (/vcl/types/string)
subfield(STRING header, STRING fieldname [, STRING separator_character])
```

Examples

```
1  if (subfield(beresp.http.Cache-Control, "private")) {
2    return (pass);
3  }
4
5  set beresp.ttl = beresp.http.Cache-Control:max-age;
6  set beresp.http.Cache-Control:max-age = "1200";
```

```
1  if (subfield(beresp.http.Set-Cookie, "httponly", ";")) {
2    #....
3  }
```

```
1  set req.http.value-of-foo = subfield(req.url.qs, "foo", "&");
```

📄 suffixof() (/vcl/functions/suffixof/)

True if the string `needle` ends the string `haystack`. An empty string is not considered a suffix.

Returns false otherwise.

Format

```
BOOL (/vcl/types/bool)
suffixof(STRING haystack, STRING needle)
```

Examples

```
1  set req.http.X-ss = std.suffixof("rectangles", "angles");;
```

📄 urldecode() (/vcl/functions/urldecode/)

Decodes a percent-encoded string. For example, `urldecode({"hello%20world+!"});` and `urldecode("hello%2520world+!");` will both return `"hello world !"`

Format

```
STRING (/vcl/types/string)
urldecode(STRING input)
```

Examples

```
1  set req.http.X-Cookie = regsub(req.url, ".*\?cookie=", ""); set req.http.Cookie = url
   decode(req.http.X-Cookie);
```

📄 urlencode() (/vcl/functions/urlencode/)

Encodes a string for use in a URL. This is also known as percent-encoding
(https://en.wikipedia.org/wiki/Percent-encoding). For example, `urlencode("hello world");` will
return `"hello%20world"`.

Format

```
STRING (/vcl/types/string)
urlencode(STRING input)
```

Examples

```
1  set req.url = req.url "?cookie=" urlencode(req.http.Cookie);
```

Miscellaneous Variables

📄 bereq.url.basename (/vcl/variables/bereq-url-basename/)

Same as `req.url.basename`, except for use between Fastly and your origin servers.

📄 bereq.url.dirname (/vcl/variables/bereq-url-dirname/)

Same as `req.url.dirname`, except for use between Fastly and your origin servers.

📄 bereq.url.qs (/vcl/variables/bereq-url-qs/)

The query string portion of `bereq.url`. This will be from immediately after the `?` to the end of the
URL.

📄 bereq.url (/vcl/variables/bereq-url/)

The URL sent to the backend. Does not include the host and scheme, meaning in
`www.example.com/index.html`, `bereq.url` would contain `/index.html`.

📄 beresp.backend.ip (/vcl/variables/beresp-backend-ip/)

The IP of the backend this response was fetched from (backported from Varnish 3).

📄 beresp.backend.name (/vcl/variables/beresp-backend-name/)

The name of the backend this response was fetched from (backported from Varnish 3).

📄 beresp.backend.port (/vcl/variables/beresp-backend-port/)

The port of the backend this response was fetched from (backported from Varnish 3).

📄 beresp.grace (/vcl/variables/beresp-grace/)

Defines how long an object can remain overdue and still have Varnish consider it for grace mode. Fastly has implemented `stale-if-error` (/guides/performance-tuning/serving-stale-content#manually-enabling-serve-stale) as a parallel implementation of `beresp.grace`.

📄 beresp.hipaa (/vcl/variables/beresp-hipaa/)

Specifies that content not be cached in non-volatile memory to help customers meet HIPAA security requirements. See our guide on HIPAA and caching PHI (/guides/compliance/hipaa-and-caching-phi) for instructions on enabling this feature for your account.

📄 beresp.pci (/vcl/variables/beresp-pci/)

Specifies that content be cached in a manner that satisfies PCI DSS requirements. See our PCI compliance description (/guides/detailed-product-descriptions/pci-compliant-caching) for instructions on enabling this feature for your account.

📄 client.port (/vcl/variables/client-port/)

Returns the remote client port. This could be useful as a seed that returns the same value both in an ESI and a top level request. For example, you could hash `client.ip` and `client.port` to get a value used both in ESI and the top level request.

📄 client.requests (/vcl/variables/client-requests/)

Tracks the number of requests received by Varnish over a persistent connection. Over an HTTP/2 connection, tracks the number of multiplexed requests.

Type

`INTEGER (/vcl/types/integer)`

Accessibility

📄 client.socket.pace (/vcl/variables/client-socket-pace/)

Ceiling rate in bytes per second for bytes sent to the client.

This rate accounts for header sizes and retransmits, so the application level rate might be different from the one set here.

Type

`INTEGER (/vcl/types/integer)`

Accessibility

Readable From

All subroutines

📄 req.grace (/vcl/variables/req-grace/)

Defines how long an object can remain overdue and still have Varnish consider it for grace mode.

📄 req.http.host (/vcl/variables/req-http-host/)

The full host name, without the path or query parameters. For example, in the request `www.example.com/index.html?a=1&b=2`, `req.http.host` will return `www.example.com`.

📄 req.is_ipv6 (/vcl/variables/req-is-ipv6/)

Indicates whether the request was made using IPv6 or not. This is a boolean, read-only variable available in `vcl_recv`, `vcl_hash`, `vcl_deliver` and `vcl_log`.

📄 req.restarts (/vcl/variables/req-restarts/)

Counts the number of times the VCL has been restarted.

📄 req.url.basename (/vcl/variables/req-url-basename/)

The file name specified in a URL. For example, in the request `www.example.com/1/hello.gif?foo=bar`, `req.url.basename` will return `hello.gif`.

📄 req.url.dirname (/vcl/variables/req-url-dirname/)

The directories specified in a URL. For example, in the request `www.example.com/1/hello.gif?foo=bar`, `req.url.dirname` will return `/1`. In the request `www.example.com/5/inner/hello.gif?foo=bar`, `req.url.dirname` will return `/5/inner`.

📄 req.url.ext (/vcl/variables/req-url-ext/)

The file extension specified in a URL. For example, in the request `www.example.com/1/hello.gif?foo=bar`, `req.url.ext` will return `gif`.

📄 req.url.path (/vcl/variables/req-url-path/)

The full path, without any query parameters. For example, in the request `www.example.com/index.html?a=1&b=2`, `req.url.path` will return `/index.html`.

📄 req.url.qs (/vcl/variables/req-url-qs/)

The query string portion of `req.url`. This will be from immediately after the `?` to the end of the URL.

📄 req.url (/vcl/variables/req-url/)

The full path, including query parameters. For example, in the request `www.example.com/index.html?a=1&b=2`, req.url will return `/index.html?a=1&b=2`.

📄 stale.exists (/vcl/variables/stale-exists/)

Specifies if a given object has stale content (/guides/performance-tuning/serving-stale-content) in cache. Returns `true` or `false`.

Query string manipulation (/vcl/query-string-manipulation/)

Examples

In your VCL, you could use `querystring.regfilter_except` as follows:

```
1   import querystring;
2
3   sub vcl_recv {
4       # return this URL with only the parameters that match this regular expression
5       set req.url = querystring.regfilter_except(req.url, "^(q|p)$");
6   }
```

You can use `querystring.regfilter` to specify a list of arguments that must not be removed (everything else will be) with a negative look-ahead expression:

```
1   set req.url = querystring.regfilter(req.url, "^(?!param1|param2)");
```

Query string manipulation Functions

📄 boltsort.sort() (/vcl/functions/boltsort-sort/)

Sorts URL parameters. For example, `boltsort.sort("/foo?b=1&a=2&c=3");` returns `"/foo?a=2&b=1&c=3"`.

Format

```
STRING (/vcl/types/string)
boltsort.sort(STRING url)
```

Examples

```
1  set req.url = boltsort.sort(req.url);
```

📄 querystring.add() (/vcl/functions/querystring-add/)

Returns the given URL with the given parameter name and value appended to the end of the query string. The parameter name and value will be URL-encoded when added to the query string.

Format

```
STRING (/vcl/types/string)
querystring.add(STRING, STRING, STRING)
```

Examples

```
1  set req.url = querystring.add(req.url, "foo", "bar");
```

📄 querystring.clean() (/vcl/functions/querystring-clean/)

Returns the given URL without empty parameters. The query-string is removed if empty (either before or after the removal of empty parameters). Note that a parameter with an empty value does not constitute an empty parameter, so a query string "?something" would not be cleaned.

Format

```
STRING (/vcl/types/string)
querystring.clean(STRING)
```

Examples

```
1  set req.url = querystring.clean(req.url);
```

📄 querystring.filter_except() (/vcl/functions/querystring-filter-except/)

Returns the given URL but only keeps the listed parameters.

Format

```
STRING (/vcl/types/string)
querystring.filter_except(STRING, STRING_LIST)
```

Examples

```
1  set req.url = querystring.filter_except(req.url,
2    "q" + querystring.filtersep() + "p");
```

📄 querystring.filter() (/vcl/functions/querystring-filter/)

Returns the given URL without the listed parameters.

Format

```
STRING (/vcl/types/string)
querystring.filter(STRING, STRING_LIST)
```

Examples

```
1  set req.url = querystring.filter(req.url,
2    "utm_source" + querystring.filtersep() +
3    "utm_medium" + querystring.filtersep() +
4    "utm_campaign");
```

📄 querystring.filtersep() (/vcl/functions/querystring-filtersep/)

Returns the separator needed by the `querystring.filter()` (/vcl/functions/querystring-filter/) and `querystring.filter_except()` (/vcl/functions/querystring-filter-except/) functions.

Format

```
STRING (/vcl/types/string)
querystring.filtersep()
```

Examples

```
1  set req.url = querystring.filter(req.url,
2    "utm_source" + querystring.filtersep() +
3    "utm_medium" + querystring.filtersep() +
4    "utm_campaign");
```

📄 querystring.globfilter_except() (/vcl/functions/querystring-globfilter-except/)

Returns the given URL but only keeps the parameters matching a glob.

Format

```
STRING (/vcl/types/string)
querystring.globfilter_except(STRING, STRING)
```

Examples

```
1  set req.url = querystring.globfilter_except(req.url, "sess*");
```

📄 querystring.globfilter() (/vcl/functions/querystring-globfilter/)

Returns the given URL without the parameters matching a glob.

Format

```
STRING (/vcl/types/string)
querystring.globfilter(STRING, STRING)
```

Examples

```
1  set req.url = querystring.globfilter(req.url, "utm_*");
```

📄 querystring.regfilter_except() (/vcl/functions/querystring-regfilter-except/)

Returns the given URL but only keeps the parameters matching a regular expression.

Format

```
STRING (/vcl/types/string)
querystring.regfilter_except(STRING, STRING)
```

Examples

```
1  set req.url = querystring.regfilter_except(req.url, "^(q|p)$");
```

📄 querystring.regfilter() (/vcl/functions/querystring-regfilter/)

Returns the given URL without the parameters matching a regular expression.

Format

```
STRING (/vcl/types/string)
querystring.regfilter(STRING, STRING)
```

Examples

```
1  set req.url = querystring.regfilter(req.url, "^utm_.*");
```

📄 querystring.remove() (/vcl/functions/querystring-remove/)

Returns the given URL with its query-string removed.

Format

```
STRING (/vcl/types/string)
querystring.remove(STRING)
```

Examples

```
1  set req.url = querystring.remove(req.url);
```

📄 querystring.set() (/vcl/functions/querystring-set/)

Returns the given URL with the given parameter name set to the given value, replacing the original value and removing any duplicates. If the parameter is not present in the query string, the parameter will be appended with the given value to the end of the query string. The parameter name and value will be URL-encoded when set in the query string.

Format

```
STRING (/vcl/types/string)
querystring.set(STRING, STRING, STRING)
```

Examples

```
1  set req.url = querystring.set(req.url, "foo", "baz");
```

📄 querystring.sort() (/vcl/functions/querystring-sort/)

Returns the given URL with its query-string sorted.

Format

```
STRING (/vcl/types/string)
querystring.sort(STRING)
```

Examples

```
1  set req.url = querystring.sort(req.url);
```

Randomness (/vcl/randomness/)

Random strings

Use the function `randomstr(length [, characters])`. When characters aren't provided, the default will be the 64 characters of `A-Za-z0-9_-`.

```
1   sub vcl_deliver {
2     set resp.http.Foo = "randomstuff=" randomstr(10);
3     set resp.http.Bar = "morsecode=" randomstr(50, ".-"); # 50 dots and dashes
4   }
```

Random content cookies in pure VCL

```
1   sub vcl_deliver {
2     add resp.http.Set-Cookie = "somerandomstuff=" randomstr(10) "; expires=" now + 18
    0d "; path=/;";
3   }
```

This adds a cookie named "somerandomstuff" with 10 random characters as value, expiring 180 days from now.

Random decisions

Use the function `randombool( _numerator_, _denominator_ )`, which has a numerator/denominator chance of returning true.

```
1   sub vcl_recv {
2     if (randombool(1, 4)) {
3       set req.http.X-AB = "A";
4     } else {
5       set req.http.X-AB = "B";
6     }
7   }
```

This will add a X-AB header to the request, with a 25% (1 out of 4) chance of having the value "A", and 75% chance of having the value "B".

The `randombool()` function accepts INT function return values, so you could do something this:

```
1   if (randombool(std.atoi(req.http.Some-Header), 100))  {
2     # do something
3   }
```

Another function, `randombool_seeded()`, takes an additional seed argument. Results for a given seed will always be the same. For instance, in this example the value of the response header will always be `no`:

```
1    if (randombool_seeded(50, 100, 12345)) {
2        set resp.http.Seeded-Value = "yes";
3    } else {
4        set resp.http.Seeded-Value = "no";
5    }
```

This could be useful for stickiness. For example, if you based the seed off of something that identified a user, you could perform A/B testing without setting a special cookie.

> ⚠ **WARNING:** The `randombool` and `randombool_seeded` functions do not use secure random numbers and should not be used for cryptographic purposes.

Randomness Functions

📄 randombool_seeded() (/vcl/functions/randombool-seeded/)

Identical to randombool (/vcl/functions/randombool/), except takes an additional parameter, which is used to seed the random number generator.

This does not use secure random numbers and should not be used for cryptographic purposes.

This function is not prefixed with the `std.` namespace.

Format

```
BOOL (/vcl/types/bool)
randombool_seeded(INTEGER numerator, INTEGER denominator, INTEGER seed)
```

Examples

```
1    set req.http.my-hmac = digest.hmac_sha256("sekrit", req.http.X-Token);
2    set req.http.hmac-chopped = regsub(req.http.my-hmac, "^(...........).*$","\1");
3    if (randombool_seeded(5,100,std.strtol(req.http.hmac-chopped ,16))) {
4        set req.http.X-Allowed = "true";
5    } else {
6        set req.http.X-Allowed = "false";
7    }
```

📄 randombool() (/vcl/functions/randombool/)

Returns a random, boolean value. The result is true when, given a pseudorandom number `r`, `(RAND_MAX * numerator) > (r * denominator)`.

This does not use secure random numbers and should not be used for cryptographic purposes.

This function is not prefixed with the `std.` namespace.

Format

```
BOOL (/vcl/types/bool)
randombool(INTEGER numerator, INTEGER denominator)
```

Examples

```
1  if (randombool(1, 10)) {
2    set req.http.X-ABTest = "A";
3  } else {
4    set req.http.X-ABTest = "B";
5  }
```

📄 randomint_seeded() (/vcl/functions/randomint-seeded/)

Identical to randomint (/vcl/functions/randomint/), except takes an additional parameter used to seed the random number generator.

This does not use secure random numbers and should not be used for cryptographic purposes.

This function is not prefixed with the `std.` namespace.

Format

```
INTEGER (/vcl/types/integer)
randomint_seeded(INTEGER from, INTEGER to, INTEGER seed)
```

Examples

```
 1  if (randomint_seeded(1, 5, user_id) < 5) {
 2    set req.http.X-ABTest = "A";
 3  } else {
 4    set req.http.X-ABTest = "B";
 5  }
 6  if (randomint_seeded(-1, 0, 555) == -1) {
 7    set req.http.X-ABTest = "A";
 8  } else {
 9    set req.http.X-ABTest = "B";
10  }
```

📄 randomint() (/vcl/functions/randomint/)

Returns a random integer value between `from` and `to`, inclusive.

This does not use secure random numbers and should not be used for cryptographic purposes.

This function is not prefixed with the `std.` namespace.

Format

```
INTEGER (/vcl/types/integer)
randomint(INTEGER from, INTEGER to)
```

Examples

```
 1  if (randomint(0, 99) < 5) {
 2    set req.http.X-ABTest = "A";
 3  } else {
 4    set req.http.X-ABTest = "B";
 5  }
 6  if (randomint(-1, 0) == -1) {
 7    set req.http.X-ABTest = "A";
 8  } else {
 9    set req.http.X-ABTest = "B";
10  }
```

📄 randomstr() (/vcl/functions/randomstr/)

Returns a random string of length `len` containing characters from the supplied string `characters`.

This does not use secure random functions and should not be used for cryptographic purposes.

This function is not prefixed with the `std.` namespace.

Format

```
STRING (/vcl/types/string)
randomstr(INTEGER len, STRING characters)
```

Examples

```
1  set req.http.X-RandomHexNum = randomstr(8, "1234567890abcdef");
```

Size (/vcl/size/)

Size Variables

📄 req.body_bytes_read (/vcl/variables/req-body-bytes-read/)

How big the body of a request was in total bytes.

Readable From

- `vcl_deliver`

- `vcl_log`

📄 req.bytes_read (/vcl/variables/req-bytes-read/)

How big a request was in total bytes.

Readable From

- `vcl_deliver`

- `vcl_log`

📄 req.header_bytes_read (/vcl/variables/req-header-bytes-read/)

How big the header of a request was in total bytes.

Readable From

All subroutines

📄 resp.body_bytes_written (/vcl/variables/resp-body-bytes-written/)

How many bytes were written for body of a response.

Readable From

- `vcl_log`

📄 resp.bytes_written (/vcl/variables/resp-bytes-written/)

How many bytes in total were sent as a response.

Readable From

- `vcl_log`

📄 resp.completed (/vcl/variables/resp-completed/)

Whether the response completed successfully or not.

Readable From

- `vcl_log`

📄 resp.header_bytes_written (/vcl/variables/resp-header-bytes-written/)

How many bytes were written for the header of a response.

Readable From

- `vcl_log`

TLS and HTTP/2 (/vcl/tls-and-http2/)

When using these variables, remember the following:

- These variables are currently only allowed to appear within the VCL hooks `vcl_recv`, `vcl_hash`, `vcl_deliver` and `vcl_log`.

- Requests made with HTTP/2 will appear in custom logs (/guides/streaming-logs/custom-log-formats) as HTTP1.1 because those requests will already have been decrypted by the time Varnish sees it. Specifically, the `%r` variable will not accurately represent the type of HTTPX request being processed.

TLS and HTTP/2 Functions

📄 h2.push() (/vcl/functions/h2-push/)

Triggers an HTTP/2 server push of the asset passed into the function as the input-string.

Format

```
VOID (/vcl/types/void)
h2.push(STRING resource)
```

Examples

```
1  if (fastly_info.is_h2 && req.url == "/") {
2    h2.push("/assets/jquery.js");
3  }
```

TLS and HTTP/2 Variables

📄 fastly_info.h2.is_push (/vcl/variables/fastly-info-h2-is-push/)

Whether or not this request was a server-initiated request generated to create an HTTP/2 Server-pushed response. Returns a boolean value.

Type

`BOOL (/vcl/types/bool)`

Readable From

- `vcl_recv`
- `vcl_hash`
- `vcl_deliver`
- `vcl_log`

📄 fastly_info.h2.stream_id (/vcl/variables/fastly-info-h2-stream-id/)

If the request was made over HTTP/2, the underlying HTTP/2 stream ID.

Readable From

- `vcl_recv`
- `vcl_hash`
- `vcl_deliver`
- `vcl_log`

📄 fastly_info.is_h2 (/vcl/variables/fastly-info-is-h2/)

Whether or not the request was made using http2.

Type

`BOOL (/vcl/types/bool)`

Readable From

- `vcl_recv`
- `vcl_hash`
- `vcl_deliver`
- `vcl_log`

📄 tls.client.cipher (/vcl/variables/tls-client-cipher/)

The cipher suite used to secure the client TLS connection. Example: `"ECDHE-RSA-AES128-GCM-SHA256"`

Readable From

- `vcl_recv`

- `vcl_hash`

- `vcl_deliver`

- `vcl_log`

📄 tls.client.ciphers_sha (/vcl/variables/tls-client-ciphers-sha/)

A SHA1 of the cipher suite identifiers sent from the client as part of the TLS handshake, represented in base64.

Readable From

- `vcl_recv`

- `vcl_hash`

- `vcl_deliver`

- `vcl_log`

📄 tls.client.protocol (/vcl/variables/tls-client-protocol/)

The TLS protocol version this connection is speaking over. Example: `"TLSv1.2"`

Readable From

- `vcl_recv`

- `vcl_hash`

- `vcl_deliver`

- `vcl_log`

📄 tls.client.servername (/vcl/variables/tls-client-servername/)

The Server Name Indication (SNI) the client sent in the `ClientHello` TLS record. Returns `""` if the client did not send SNI. Returns `NULL` (the undefined string) if the request is not a TLS request.

Readable From

- `vcl_recv`

- `vcl_hash`

- `vcl_deliver`

- `vcl_log`

📄 tls.client.tlsexts_sha (/vcl/variables/tls-client-tlsexts-sha/)

A SHA1 of the TLS extension identifiers sent from the client as part of the TLS handshake, represented in base64.

Readable From

- `vcl_recv`

- `vcl_hash`

- `vcl_deliver`

- `vcl_log`

UUID (/vcl/uuid/)

UUID Functions

📄 uuid.dns() (/vcl/functions/uuid-dns/)

Returns the [RFC4122 (https://tools.ietf.org/html/rfc4122)](https://tools.ietf.org/html/rfc4122) identifier of DNS namespace, namely the constant `"6ba7b810-9dad-11d1-80b4-00c04fd430c8"`.

Format

```
STRING (/vcl/types/string)
uuid.dns()
```

Examples

```
1   declare local var.dns STRING;
2   set var.dns = uuid.version3(uuid.dns(), "www.example.com");
3   # var.dns now contains "5df41881-3aed-3515-88a7-2f4a814cf09e"
```

📄 uuid.is_valid() (/vcl/functions/uuid-is-valid/)

Returns true if the string holds a textual representation of a valid UUID (per [RFC4122 (https://tools.ietf.org/html/rfc4122)](https://tools.ietf.org/html/rfc4122)). False otherwise.

Format

```
BOOL (/vcl/types/bool)
uuid.is_valid(STRING string)
```

Examples

```
1  if (uuid.is_valid(req.http.X-Unique-Id)) {
2    set beresp.http.X-Unique-Id-Valid = "yes";
3  }
```

📄 uuid.is_version3() (/vcl/functions/uuid-is-version3/)

Returns true if string holds a textual representation of a valid version 3 UUID. False otherwise.

Format

```
BOOL (/vcl/types/bool)
uuid.is_version3(STRING string)
```

Examples

```
1  if (uuid.is_version3(req.http.X-Unique-Id)) {
2    set beresp.http.X-Unique-Id-Valid-V3 = "yes";
3  }
```

📄 uuid.is_version4() (/vcl/functions/uuid-is-version4/)

Returns true if string holds a textual representation of a valid version 4 UUID. False otherwise.

Format

```
BOOL (/vcl/types/bool)
uuid.is_version4(STRING string)
```

Examples

```
1  if (uuid.is_version4(req.http.X-Unique-Id)) {
2    set beresp.http.X-Unique-Id-Valid-V4 = "yes";
3  }
```

📄 uuid.is_version5() (/vcl/functions/uuid-is-version5/)

Returns true if string holds a textual representation of a valid version 5 UUID. False otherwise.

Format

```
BOOL (/vcl/types/bool)
uuid.is_version5(STRING string)
```

Examples

```
1  if (uuid.is_version5(req.http.X-Unique-Id)) {
2    set beresp.http.X-Unique-Id-Valid-V5 = "yes";
3  }
```

📄 uuid.oid() (/vcl/functions/uuid-oid/)

Returns the RFC4122 (https://tools.ietf.org/html/rfc4122) identifier of ISO OID namespace, namely the constant `"6ba7b812-9dad-11d1-80b4-00c04fd430c8"`.

Format

```
STRING (/vcl/types/string)
uuid.oid()
```

Examples

```
1  declare local var.oid STRING;
2  set var.oid = uuid.version3(uuid.oid(), "2.999");
3  # var.oid now contains "31cb1efa-18c4-3d19-89ba-df6a74ddbd1d"
```

📄 uuid.url() (/vcl/functions/uuid-url/)

Returns the RFC4122 (https://tools.ietf.org/html/rfc4122) identifier of URL namespace, namely the constant `"6ba7b811-9dad-11d1-80b4-00c04fd430c8"`.

Format

```
STRING (/vcl/types/string)
uuid.url()
```

Examples

```
1  declare local var.url STRING;
2  set var.url = uuid.version3(uuid.url(), "https://www.example.com/");
3  # var.url now contains "7fed185f-0864-319f-875b-a3d5458e30ac"
```

📄 uuid.version3() (/vcl/functions/uuid-version3/)

Derives a UUID corresponding to `name` within the given `namespace` using MD5 hash function. Namespace itself is identified by a UUID. Name must be in a canonical form appropriate for selected namespace.

> ℹ️ **NOTE:** In principle, names can be arbitrary octet strings. This implementation will, however, truncate at the first NUL byte.

Format

```
STRING (/vcl/types/string)
uuid.version3(STRING namespace, STRING name)
```

Examples

```
1   set req.http.X-Unique-Id = uuid.version3(uuid.dns(), "www.fastly.com");
```

📄 uuid.version4() (/vcl/functions/uuid-version4/)

Returns a UUID based on random number generator output.

Format

```
STRING (/vcl/types/string)
uuid.version4()
```

Examples

```
1   set req.http.X-Unique-Id = uuid.version4();
```

📄 uuid.version5() (/vcl/functions/uuid-version5/)

Derives a UUID corresponding to `name` within the given `namespace` using SHA1 hash function. Namespace itself is identified by a UUID. Name must be in a canonical form appropriate for selected namespace.

> ℹ️ **NOTE:** In principle, names can be arbitrary octet strings. This implementation will, however, truncate at the first NUL byte.

Format

```
STRING (/vcl/types/string)
uuid.version5(STRING namespace, STRING name)
```

Examples

```
1   set req.http.X-Unique-Id = uuid.version5(uuid.dns(), "www.fastly.com");
```

📄 uuid.x500() (/vcl/functions/uuid-x500/)

Returns the RFC4122 (https://tools.ietf.org/html/rfc4122) identifier of X.500 namespace, namely the constant `"6ba7b812-9dad-11d1-80b4-00c04fd430c8"`.

Format

```
STRING (/vcl/types/string)
uuid.x500()
```

Examples

```
1  declare local var.x500 STRING;
2  set var.x500 = uuid.version3(uuid.x500(), "CN=Test User 1, O=Example Organization, ST
   =California, C=US");
3  # var.x500 now contains "addf5e97-9287-3834-abfd-7edcbe7db56f"
```

Guides

§ Custom VCL

📄 Creating custom VCL (/vcl/custom-vcl/creating-custom-vcl/)

Fastly Varnish syntax is specifically compatible with Varnish 2.1.5 (https://varnish-cache.org/docs/2.1). We run a custom version with added functionality and our VCL parser has its own pre-processor. To mix and match Fastly VCL with your custom VCL successfully, remember the following:

- **You can only restart Varnish requests three times.** This limit exists to prevent infinite loops.

- **VCL doesn't take kindly to Windows newlines (line breaks).** It's best to avoid them entirely.

- **It's best to use** `curl -X PURGE` **to initiate purges via API (/api/purge).** To restrict access to purging, check for the `FASTLYPURGE` method not the `PURGE` method. When you send a request to Varnish to initiate a purge, the HTTP method that you use is "PURGE", but it has already been changed to "FASTLYPURGE" by the time your VCL runs that request.

- **If you override TTLs with custom VCL, your default TTL set in the configuration (/guides/performance-tuning/serving-stale-content) will not be honored** and the expected behavior may change.

Inserting custom VCL in Fastly's VCL boilerplate

> ⚠️ **DANGER:** Include all of the Fastly VCL boilerplate as a template in your custom VCL file, especially the VCL macro lines (they start with `#FASTLY`). VCL macros expand the code into generated VCL. Add your custom code *in between* the different sections as shown in the example unless you specifically intend to override the VCL at that point.

Custom VCL placement example

```
1    sub vcl_miss {
2      # my custom code
3      if (req.http.User-Agent ~ "Googlebot") {
4        set req.backend = F_special_google_backend;
5      }
6    #FASTLY miss
7      return(fetch);
8    }
```

Fastly's VCL boilerplate

> ⭐ **TIP:** If you use the Fastly Image Optimizer, use the image optimization VCL boilerplate (/guides/imageopto-setup-use/image-optimization-vcl-boilerplate) instead.

```vcl
1   sub vcl_recv {
2   #FASTLY recv
3
4     if (req.method != "HEAD" && req.method != "GET" && req.method != "FASTLYPURGE")
5   {
6       return(pass);
7     }
8
9     return(lookup);
10  }
11
12  sub vcl_fetch {
13  #FASTLY fetch
14
15    if ((beresp.status == 500 || beresp.status == 503) && req.restarts < 1 && (req.m
16  ethod == "GET" || req.method == "HEAD")) {
17      restart;
18    }
19
20    if (req.restarts > 0) {
21      set beresp.http.Fastly-Restarts = req.restarts;
22    }
23
24    if (beresp.http.Set-Cookie) {
25      set req.http.Fastly-Cachetype = "SETCOOKIE";
26      return(pass);
27    }
28
29    if (beresp.http.Cache-Control ~ "private") {
30      set req.http.Fastly-Cachetype = "PRIVATE";
31      return(pass);
32    }
33
34    if (beresp.status == 500 || beresp.status == 503) {
35      set req.http.Fastly-Cachetype = "ERROR";
36      set beresp.ttl = 1s;
37      set beresp.grace = 5s;
38      return(deliver);
39    }
40
41    if (beresp.http.Expires || beresp.http.Surrogate-Control ~ "max-age" || beresp.h
42  ttp.Cache-Control ~ "(s-maxage|max-age)") {
43      # keep the ttl here
44    } else {
45      # apply the default ttl
46      set beresp.ttl = 3600s;
47    }
48
49    return(deliver);
50  }
51
52  sub vcl_hit {
53  #FASTLY hit
```

```
54
55    if (!obj.cacheable) {
56        return(pass);
57    }
58    return(deliver);
59  }
60
61  sub vcl_miss {
62  #FASTLY miss
63    return(fetch);
64  }
65
66  sub vcl_deliver {
67  #FASTLY deliver
68    return(deliver);
69  }
70
71  sub vcl_error {
72  #FASTLY error
73  }
74
75  sub vcl_pass {
76  #FASTLY pass
77  }
78

    sub vcl_log {
    #FASTLY log
    }
```

📄Uploading custom VCL (/vcl/custom-vcl/uploading-custom-vcl/)

Fastly allows you create your own Varnish Configuration Language (VCL) files with specialized configurations. By uploading custom VCL files, you can use custom VCL and Fastly VCL together at the same time (/vcl/custom-vcl/creating-custom-vcl/). Keep in mind that your custom VCL always takes precedence over VCL generated by Fastly.

> ❶ **IMPORTANT:** Personal data should not be incorporated into VCL. Our Compliance and Law FAQ (/guides/compliance-and-law-faq/) describes in detail how Fastly handles personal data privacy.

Uploading a VCL file

Follow these instructions to upload a custom VCL file:

1. Log in to the Fastly web interface and click the **Configure** link.

2. From the service menu, select the appropriate service.

3. Click the **Configuration** button and then select **Clone active**. The Domains page appears.

4. Click the **Custom VCL** tab. The Custom VCL page appears.

5. Click the **Upload a new VCL file** button. The Upload a new VCL file page appears.

6. In the **Name** field, enter the name of the VCL file. For included files, this name must match the include statement in the main VCL file. See how to include additional VCL configurations for more information.

7. Click **Upload file** and select a file to upload. The name of the uploaded file appears next to the button.

> ❗ **IMPORTANT:** Don't upload generated VCL that you've downloaded from the Fastly web interface. Instead, edit and then upload a copy of Fastly's VCL boilerplate (/vcl/custom-vcl/creating-custom-vcl#fastlys-vcl-boilerplate) to avoid errors.

8. Click the **Create** button. The VCL file appears in the Varnish Configurations area.

9. Click the **Activate** button to deploy your configuration changes.

Editing a VCL file

To edit an existing VCL file, follow these instructions:

1. Log in to the Fastly web interface and click the **Configure** link.

2. From the service menu, select the appropriate service.

3. Click the **Configuration** button and then select **Clone active**. The Domains page appears.

4. Click the **Custom VCL** tab. The Custom VCL page appears.

5. In the **Varnish Configurations** area, click the VCL file you want to edit. The Edit an existing VCL file page appears.

Edit an existing VCL file

Our VCL tutorial will help you get started with creating VCL files.

Name   My custom VCL   ★ Required

For included files, this name must exactly match the include statement in the main VCL file.

Existing file   View Source   Download

Config file   REPLACE FILE

UPDATE   CANCEL

6. In the **Name** field, optionally enter a new name of the VCL file.

7. Click the **Download** link to download the appropriate file.

8. Make the necessary changes to your file and save them.

9. Click the **Replace file** button and select the file you updated. The selected file replaces the current VCL file and the file name appears next to the button.

10. Click the **Update** button to update the VCL file in the Fastly application.

11. Click the **Activate** button to deploy your configuration changes.

Including additional VCL configurations

You can apply additional VCL files along with your main VCL by including their file names in the main VCL file using the syntax `include "VCL Name"` where `VCL Name` is the name of an included VCL object you've created.

For example, if you've created an included VCL object called "ACL" (to use an access control list (/guides/access-control-lists/manually-creating-access-control-lists) for code manageability) and the file is named `acl.vcl`, your main VCL configuration file would need to contain this line:

```
include "ACL"
```

📄Previewing and testing VCL (/vcl/custom-vcl/previewing-and-testing-vcl/)

Any time you upload VCL files (/vcl/custom-vcl/uploading-custom-vcl/) you can preview and test the VCL prior to activating a new version of your service.

Previewing VCL before activation

To preview VCL prior to activating a service version.

1. Log in to the Fastly web interface and click the **Configure** link.

2. From the service menu, select the appropriate service.

3. Click the **Configuration** button and then select **Clone active**. The Domains page appears.

4. Click the **Options** button to open the **Manage version** menu and select **Show VCL**.
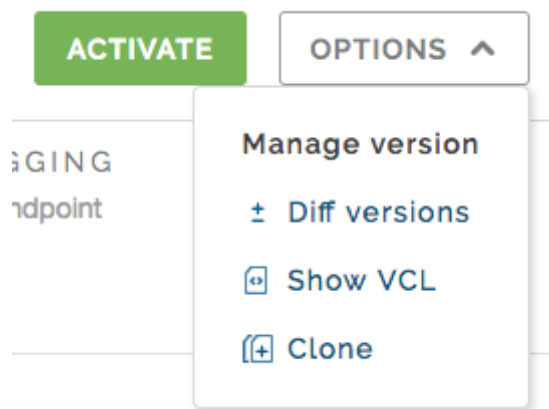
The VCL preview page appears.

Testing VCL configurations

You don't need a second account to test your VCL configurations. We recommend adding a new service within your existing account that's specifically designed for testing. A name like "QA" or "testing" or "staging" makes distinguishing between services much easier.

Once created, simply point your testing service to your testing or QA environment. Edit your Fastly configurations for the testing service as if you were creating them for production. Preview your VCL, test things out, and tweak them to get them perfect.

When your testing is complete, make the same changes in your production service that you made to your testing service. If you are using custom VCL, upload the VCL file (/vcl/custom-vcl/uploading-custom-vcl/) to the production service you'll be using.

§ VCL Snippets

📄About VCL Snippets (/vcl/vcl-snippets/about-vcl-snippets/)

VCL Snippets are short blocks of VCL logic (/guides/vcl-tutorials/guide-to-vcl) that can be included directly in your service configurations. They're ideal for adding small sections of code when you don't need more complex, specialized configurations that sometimes require custom VCL (/vcl/custom-vcl/uploading-custom-vcl/). Fastly supports two types of VCL Snippets:

- **Regular VCL Snippets (/vcl/vcl-snippets/using-regular-vcl-snippets/)** get created as you create versions of your Fastly configurations. They belong to a specific service and any modifications you make to the snippet are locked and deployed when you deploy a new version of that service. You can treat regular snippets like any other Fastly objects because we continue to clone them and deploy them with a service until you specifically delete them. You can create regular snippets using either the web interface or via the API.

- **Dynamic VCL Snippets (/guides/vcl-snippets/using-dynamic-vcl-snippets)** can be modified and deployed any time they're changed. Because they are versionless objects (much like Edge Dictionaries (/guides/edge-dictionaries/) or ACLs (/guides/access-control-lists/) at the edge), dynamic snippets can be modified independently from service changes. This means you can modify snippet code rapidly without deploying a service version that may not be ready for production. You can only create dynamic snippets via the API.

Limitations of VCL Snippets

- Snippets are limited to 1MB in size by default. If you need to store snippets larger than the limit, contact support@fastly.com (mailto:support@fastly.com).

- Snippets do not currently support conditions, though `if` statements can be used within snippet code instead.

- Snippets cannot currently be shared between services.

📄Using dynamic VCL Snippets (/vcl/vcl-snippets/using-dynamic-vcl-snippets/)

Dynamic VCL Snippets are one of two types of snippets (/vcl/vcl-snippets/about-vcl-snippets) that allow you to insert small sections of VCL logic into your service configuration without requiring custom VCL (/vcl/custom-vcl/uploading-custom-vcl/) (though you can still include snippets in custom VCL when necessary).

You can only create dynamic snippets via the API. Because they are versionless objects (much like Edge Dictionaries (/guides/edge-dictionaries/) or ACLs (/guides/access-control-lists/) at the edge), dynamic snippets can be modified independently from changes to your Fastly service. This means you can modify snippet code rapidly without deploying a service version that may not be ready for production.

Creating and using a dynamic VCL Snippet

Using the cURL command line tool, make the following API call in a terminal application:

```
1  curl -X POST -s https://api.fastly.com/service/<Service ID>/version/<Editable Version #>/snippet -H "Fastly-Key:FASTLY_API_TOKEN" -H 'Content-Type: application/x-www-form-urlencoded' --data $'name=my_dynamic_snippet_name&type=recv&dynamic=1&content=if ( req.url ) {\n set req.http.my-snippet-test-header = "true";\n}';
```

Fastly returns a JSON response that looks like this:

```
 1  {
 2    "service_id": "<Service Id>",
 3    "version": "<Editable Version>",
 4    "name": "my_dynamic_snippet_name",
 5    "type": "recv",
 6    "priority": 100,
 7    "dynamic": 1,
 8    "content": null,
 9    "id": "decafbad12345",
10    "created_at": "2016-09-09T20:34:51+00:00",
11    "updated_at": "2016-09-09T20:34:51+00:00",
12    "deleted_at": null
13  }
```

> ❶ **NOTE:** The returned JSON includes `"content": null`. This happens because the content is stored in a separate, unversioned object.

Viewing dynamic VCL Snippets in the web interface

You can view a list of dynamic VCL snippets. You can also view just the source of a specific snippet or a specific snippet's location in generated VCL.

Viewing a list of dynamic VCL Snippets

To view the entire list of a service's dynamic VCL Snippets directly in the web interface:

1. Log in to the Fastly web interface and click the **Configure** link.

2. From the service menu, select the appropriate service.

3. Click the **VCL Snippets** link. The VCL Snippets page appears listing all dynamic VCL Snippets for your service in the Dynamic snippets area.

Viewing the source of a specific snippet

You can view just the source of a specific snippet:

1. Log in to the Fastly web interface and click the **Configure** link.

2. From the service menu, select the appropriate service.

3. Click the **VCL Snippets** link. The VCL Snippets page appears.

4. Click the **View Source** link to the right of the name of the snippet. A view source window appears.

Viewing the location of a specific snippet in generated VCL

You can view a specific snippet's location in generated VCL:

1. Log in to the Fastly web interface and click the **Configure** link.

2. From the service menu, select the appropriate service.

3. Click the **VCL Snippets** link. The VCL Snippets page appears.

4. Click the **Show in Generated VCL** link to the right of the name of the snippet. The Generated VCL window appears.

Fetching a list of all dynamic VCL Snippets

To list all dynamic VCL Snippets attached to a service, make the following API call in a terminal application:

```
1   curl -X GET -s https://api.fastly.com/service/<Service ID>/version/<Editable Versio
    n #>/snippet -H "Fastly-Key:FASTLY_API_TOKEN"
```

Fetching an individual dynamic VCL Snippet

To fetch an individual snippet, make the following API call in a terminal application:

```
1   curl -X GET -s https://api.fastly.com/service/<Service ID>/snippet/<my_dynamic_snip
    pet_id> -H "Fastly-Key:FASTLY_API_TOKEN"
```

Unlike [fetching regular VCL Snippets (/vcl/vcl-snippets/using-regular-vcl-snippets#fetching-an-individual-regular-vcl-snippet)](/vcl/vcl-snippets/using-regular-vcl-snippets#fetching-an-individual-regular-vcl-snippet), you do not include the version in the URL and you must use the ID returned when the snippet was created, not the name.

Updating an existing dynamic VCL Snippet

To update an individual snippet, make the following API call in a terminal application:

```
1   curl -X PUT -s https://api.fastly.com/service/<Service ID>/snippet/<my_dynamic_snip
    pet_id> -H "Fastly-Key:FASTLY_API_TOKEN" -H 'Content-Type: application/x-www-form-u
    rlencoded' --data $'content=if ( req.url ) {\n set req.http.my-snippet-test-header
    = \"affirmative\";\n}';
```

Deleting an existing dynamic VCL Snippet

To delete an individual snippet, make the following API call in a terminal application:

```
1   curl -X DELETE -s https://api.fastly.com/service/<Service ID>/version/<Editable Ver
    sion #>/snippet/<my_dynamic_snippet_name> -H "Fastly-Key:FASTLY_API_TOKEN"
```

Including dynamic snippets in custom VCL

By specifying a location of `none` for the `type` parameter, snippets will not be rendered in VCL. This allows you to include snippets in custom VCL using the following syntax:

`include "snippet::<snippet name>"`

The same VCL Snippet can be included in custom VCL in as many places as needed.

Example use: blocking site scrapers

Say you wanted to implement some pattern matching against incoming requests to block someone trying to scrape your site. Say also that you've developed a system that looks at all incoming requests and generates a set of rules that can identify scrapers using a combination of the incoming IP address, the browser, and the URL they're trying to fetch. Finally, say that the system updates the rules every 20 minutes.

If, during system updates, your colleagues are also making changes to the rest of your Fastly configuration, you probably don't want the system to automatically deploy the latest version of the service since it might be untested. Instead you could generate the rules as a Dynamic VCL Snippet. Whenever the snippet is updated, all other logic remains the same as the currently deployed version and only your rules are modified.

📄Using regular VCL Snippets (/vcl/vcl-snippets/using-regular-vcl-snippets/)

Regular VCL Snippets are one of <u>two types of snippets (/vcl/vcl-snippets/about-vcl-snippets)</u> that allow you to insert small sections of VCL logic into your service configuration without requiring <u>custom VCL (/vcl/custom-vcl/uploading-custom-vcl/)</u> (though you can still include snippets in custom VCL when necessary).

Unlike <u>dynamic snippets (/vcl/vcl-snippets/using-dynamic-vcl-snippets)</u>, regular snippets can be created via the web interface or via the API. They are considered "versioned" objects. They belong to a specific service and any modifications you make to the snippet are locked and deployed when you deploy a new version of that service. We continue to clone them and deploy them with a service until you specifically delete them.

Creating a regular VCL Snippet

You can create regular VCL Snippets via the web interface or via the API.

Via the web interface

To create a regular VCL Snippet via the web interface:

1. Log in to the Fastly web interface and click the **Configure** link.

2. From the service menu, select the appropriate service.

3. Click the **VCL Snippets** link. The VCL Snippets page appears.

4. Click **Create Snippet**. The Create a VCL snippet page appears.

Create a VCL snippet

Name [   ]   ✱ Required

Type (placement of the snippet)

This **specifies the location** in which to place the snippet

- ● **init** - inserts the snippets *above* all subroutines (good for defining backends, access control lists, tables)

- ○ **within subroutine** - inserts the snippets *within* a subroutine (following any boilerplate code and preceeding any objects)

  [Select subroutine… ⬍]

- ○ **none (advanced)** - requires you to manually insert the snippet using custom VCL

VCL

[   ]

> Advanced option   Priority

[**CREATE**]   CANCEL

5. In the **Name** field, type an appropriate name (for example, `Example Snippet`).

6. Using the **Type** controls, select the location in which the snippet should be placed as follows:
   - Select `init` to insert it above all subroutines in your VCL.

- Select `within subroutine` to insert it within a specific subroutine and then select the specific subroutine from the **Select subroutine** menu.

- Select `none (advanced)` to insert it manually. See Including regular snippets in custom VCL (/vcl/vcl-snippets/using-regular-vcl-snippets#including-regular-snippets-in-custom-vcl) for the additional manual insertion requirements if you select this option.

7. In the **VCL** field, type the snippet of VCL logic to be inserted for your service version.

8. Click **Create** to create the snippet.

Via the API

To create a regular VCL Snippet via the API, make the following API call using the cURL command line tool in a terminal application:

```
1   curl -X POST -s https://api.fastly.com/service/<Service ID>/version/<Editable Versi
    on #>/snippet -H "Fastly-Key:FASTLY_API_TOKEN" -H `fastly-cookie` -H 'Content-Type:
    application/x-www-form-urlencoded' --data $'name=my_regular_snippet&type=recv&dyna
    mic=0&content=if ( req.url ) {\n set req.http.my-snippet-test-header = "true";\n}';
```

Fastly returns a JSON response that looks like this:

```
1   {
2     "service_id": "<Service Id>",
3     "version": "<Editable Version>",
4     "name": "my_regular_snippet",
5     "type": "recv",
6     "content": "if ( req.url ) {\n set req.http.my-snippet-test-header = \"true\";\n
7   }",
8     "priority": 100,
9     "dynamic": 0,
10    "id": "56789exampleid",
11    "created_at": "2016-09-09T20:34:51+00:00",
12    "updated_at": "2016-09-09T20:34:51+00:00",
13    "deleted_at": null
    }
```

> ℹ **NOTE:** When regular VCL snippets get created, an `id` field will be returned that isn't used. The field only applies to dynamic VCL Snippets (/vcl/vcl-snippets/using-dynamic-vcl-snippets). In addition, the returned JSON includes a populated `content` field because the snippet content is stored in a versioned object.

Viewing regular VCL Snippets in the web interface

You can view a list of regular VCL snippets. You can also view just the source of a specific snippet or a specific snippet's location in generated VCL.

Viewing a list of regular VCL Snippets

To view the entire list of a service's regular VCL Snippets directly in the web interface:

1. Log in to the Fastly web interface and click the **Configure** link.

2. From the service menu, select the appropriate service.

3. Click the **VCL Snippets** link. The VCL Snippets page appears listing all available VCL snippets for your service.

Viewing the source of a specific snippet

You can view just the source of a specific snippet:

1. Log in to the Fastly web interface and click the **Configure** link.

2. From the service menu, select the appropriate service.

3. Click the **VCL Snippets** link. The VCL Snippets page appears.

4. Click the **View Source** link to the right of the name of the snippet. A view source window appears.

Viewing the location of a specific snippet in generated VCL

You can view a specific snippet's location in generated VCL:

1. Log in to the Fastly web interface and click the **Configure** link.

2. From the service menu, select the appropriate service.

3. Click the **VCL Snippets** link. The VCL Snippets page appears.

4. Click the **Show in Generated VCL** link to the right of the name of the snippet. The Generated VCL window appears.

Fetching regular VCL Snippets via the API

You can fetch regular VCL Snippets for a particular service via the API either singly or all at once.

Fetching an individual regular VCL Snippet

To fetch an individual snippet, make the following API call in a terminal application:

```
1   curl -X GET -s https://api.fastly.com/service/<Service ID>/version/<Editable Versio
    n #>/snippet/<Snippet Name e.g my_regular_snippet> -H "Fastly-Key:FASTLY_API_TOKEN"
```

Unlike fetching dynamic VCL Snippets (/vcl/vcl-snippets/using-dynamic-vcl-snippets#fetching-an-individual-dynamic-vcl-snippet) you include the version in the URL and you must use the name of the snippet, not the ID.

Fetching a list of regular VCL Snippets

To list all regular VCL Snippets attached to a service, make the following API call in a terminal application:

```
1   curl -X GET -s https://api.fastly.com/service/<Service ID>/version/<Editable Versio
    n #>/snippet/ -H "Fastly-Key:FASTLY_API_TOKEN"
```

Updating an existing regular VCL Snippet

You can update existing regular VCL Snippets via the web interface or via the API.

Via the web interface

To update an individual snippet via the web interface:

1. Log in to the Fastly web interface and click the **Configure** link.

2. From the service menu, select the appropriate service.

3. Click the **VCL Snippets** link. The VCL Snippets page appears.

4. Click the pencil icon next to the name of the snippet to be updated.

Example Snippet ✏️   View Source   Show in Generated VCL   🗑️
Priority: 100
Type: init

The Edit snippet page appears.

Edit snippet

VCL snippet guide

Name

Example Snippet

★ Required

Type (placement of the snippet)   This specifies the location in which to place the snippet

○ **init** - inserts the snippets *above* all subroutines (good for defining backends, access control lists, tables)

○ **within subroutine** - inserts the snippets *within* a subroutine (following any boilerplate code and preceeding any objects)

○ **none (advanced)** - requires you to manually insert the snippet using custom VCL

VCL

```
Example Snippet VCL
```

❯ Advanced option   Priority

UPDATE   CANCEL

5. Update the snippet's settings or VCL as appropriate.

6. Click **Update** to save your changes.

Via the API

To update an individual snippet via the API, make the following API call in a terminal application:

```
1    curl -X PUT -s https://api.fastly.com/service/<Service ID>/version/<Editable Versio
     n #>/snippet/<Snippet Name e.g my_regular_snippet> -H "Fastly-Key:FASTLY_API_TOKEN"
      -H 'Content-Type: application/x-www-form-urlencoded' --data $'content=if ( req.url
     ) {\n set req.http.my-snippet-test-header = \"affirmative\";\n}';
```

Deleting an existing regular VCL Snippet

You can update existing regular VCL Snippets via the web interface or via the API.

Via the web interface

1. Log in to the Fastly web interface and click the **Configure** link.

2. From the service menu, select the appropriate service.

3. Click the **VCL Snippets** link. The VCL Snippets page appears.

4. Click the trashcan icon to the right of the name of the snippet to be updated.

A confirmation window appears.

5. Click **Confirm and Delete**.

Via the API

To delete an individual snippet via the API, make the following API call in a terminal application:

```
1   curl -X DELETE -s https://api.fastly.com/service/<Service ID>/version/<Editable Ver
    sion #>/snippet/<Snippet Name e.g my_regular_snippet> -H "Fastly-Key:FASTLY_API_TOK
    EN"
```

Including regular snippets in custom VCL

Snippets will not be rendered in VCL if you select `none (advanced)` for the snippet type in the web interface or specify a location of `none` for the `type` parameter in the API. This allows you to manually include snippets in custom VCL using the following syntax:

`include "snippet::<snippet name>"`

The same VCL Snippet can be included in custom VCL in as many places as needed.

Example use: location-based redirection

Say that you work at a large content publisher and you want to redirect users to different editions of your publication depending on which country their request comes from. Say also that you want the ability to override the edition you deliver to them based on a cookie.

Using regular VCL snippets, you could add a new object with the relevant VCL as follows:

```
1    if (req.http.Cookie:edition == "US" || client.geo.country_code == "US" || ) {
2      set req.http.Edition = "US";
3      set req.backend = F_US;
4    } elseif (req.http.Cookie:edition == "Europe" || server.region ~ "^EU-" ) {
5      set req.http.Edition = "EU";
6      set req.backend = F_European;
7    } else {
8      set req.http.Edition = "INT";
9      set req.backend = F_International;
10   }
```

This would create an Edition header in VCL, but allow you to override it by setting a condition. You would add the Edition header into Vary (https://www.fastly.com/blog/best-practices-using-vary-header) and then add a false condition (/guides/conditions/using-conditions#using-operators-to-perform-matches-on-complex-logical-expressions) (e.g., `!reg.url`) to your other backends to ensure the correct edition of your publication gets delivered (Remember: VCL Snippets get added to VCL before backends are set.)

§ VCL Reference

📄 Functions (/vcl/functions/)

These VCL functions are supported by Fastly.

Cryptographic (/vcl/cryptographic/)

Fastly provides several functions in VCL (/guides/vcl-tutorials/) for cryptographic- and hashing-related purposes. It is based very heavily on Kristian Lyngstøl's digest vmod (https://github.com/varnish/libvmod-digest) for Varnish 3 (which means you can also refer to that documentation for more detail).

- digest.awsv4_hmac() (/vcl/functions/digest-awsv4-hmac/) — Returns an AWSv4 message authentication code based on the supplied `key` and `string`.

- digest.base64_decode() (/vcl/functions/digest-base64-decode/) — Decode Base64.

- digest.base64() (/vcl/functions/digest-base64/) — Base64 encoding.

- digest.base64url_decode() (/vcl/functions/digest-base64url-decode/) — Decode Base64 with url safe characters in.

- digest.base64url_nopad_decode() (/vcl/functions/digest-base64url-nopad-decode/) — Decode Base64 with url safe characters.

- digest.base64url_nopad() (/vcl/functions/digest-base64url-nopad/) — Base64 encoding.

- digest.base64url() (/vcl/functions/digest-base64url/) — Base64 encoding.

- digest.hash_crc32() (/vcl/functions/digest-hash-crc32/) — Use a 32 bit Cyclic Redundancy Checksum.

- digest.hash_crc32b() (/vcl/functions/digest-hash-crc32b/) — A reversed CRC32 (for compatibility with some PHP applications).

- digest.hash_md5() (/vcl/functions/digest-hash-md5/) — Use the MD5 hash.

- digest.hash_sha1() (/vcl/functions/digest-hash-sha1/) — Use the SHA1 hash.

- digest.hash_sha224() (/vcl/functions/digest-hash-sha224/) — Use the SHA224 hash.

- digest.hash_sha256() (/vcl/functions/digest-hash-sha256/) — Use the SHA256 hash.

- digest.hash_sha384() (/vcl/functions/digest-hash-sha384/) — Use the SHA384 hash.

- digest.hash_sha512() (/vcl/functions/digest-hash-sha512/) — Use the SHA512 hash.

- digest.hmac_md5_base64() (/vcl/functions/digest-hmac-md5-base64/) — Hash-based message authentication code using MD5.

- digest.hmac_md5() (/vcl/functions/digest-hmac-md5/) — Hash-based message authentication code using MD5.

- digest.hmac_sha1_base64() (/vcl/functions/digest-hmac-sha1-base64/) — Hash-based message authentication code using SHA-1.

- digest.hmac_sha1() (/vcl/functions/digest-hmac-sha1/) — Hash-based message authentication code using SHA-1.

- digest.hmac_sha256_base64() (/vcl/functions/digest-hmac-sha256-base64/) — Hash-based message authentication code using SHA-256.

- digest.hmac_sha256() (/vcl/functions/digest-hmac-sha256/) — Hash-based message authentication code using SHA-256.

- digest.rsa_verify() (/vcl/functions/digest-rsa-verify/) — A boolean function that returns true if the RSA signature of `payload` using `public_key` matches `digest`.

- digest.secure_is_equal() (/vcl/functions/digest-secure-is-equal/) — A boolean function that returns true if s1 and s2 are equal.

- digest.time_hmac_md5() (/vcl/functions/digest-time-hmac-md5/) — Time based One Time Password using MD5.

- digest.time_hmac_sha1() (/vcl/functions/digest-time-hmac-sha1/) — Time based One Time Password using SHA1.

- digest.time_hmac_sha256() (/vcl/functions/digest-time-hmac-sha256/) — Time based One Time Password using SHA256.

Date and time (/vcl/date-and-time/)

By default VCL includes the `now` variable, which provides the current time (for example, `Mon, 02 Jan 2006 22:04:05 GMT`). Fastly adds several new Varnish variables and functions that allow more flexibility when dealing with dates and times.

- std.integer2time() (/vcl/functions/std-integer2time/) — Converts an integer, representing seconds since the UNIX Epoch, to a time variable.

- std.time() (/vcl/functions/std-time/) — Converts a string to a time variable.

- strftime() (/vcl/functions/strftime/) — Formats a time to a string.

- time.add() (/vcl/functions/time-add/) — Adds a relative time to a time.

- time.hex_to_time() (/vcl/functions/time-hex-to-time/) — This specialized function takes a hexadecimal string value, divides by `divisor` and interprets the result as seconds since the

UNIX Epoch.

- time.is_after() (/vcl/functions/time-is-after/) — Returns true if `t1` is after `t2`.

- time.sub() (/vcl/functions/time-sub/) — Subtracts a relative time from a time.

Miscellaneous (/vcl/miscellaneous/)

Fastly has added several miscellaneous features to Varnish that don't easily fit into specific categories.

- cstr_escape() (/vcl/functions/cstr-escape/) — Escapes bytes unsafe for printing from a string using C-style escape sequences.

- http_status_matches() (/vcl/functions/http-status-matches/) — Determines whether or not an HTTP status code matches a pattern.

- if() (/vcl/functions/if/) — Implements a ternary operator for strings; if the expression is true, it returns `value-when-true`; if the expression is false, it returns `value-when-false`.

- json.escape() (/vcl/functions/json-escape/) — Escapes characters of a UTF-8 encoded Unicode string using JSON-style escape sequences.

- prefixof() (/vcl/functions/prefixof/) — True if the string `haystack` begins with the string `needle`.

- regsub() (/vcl/functions/regsub/) — Replaces the first occurrence of `pattern`, which may be a Perl-compatible regular expression, in `input` with `replacement`.

- regsuball() (/vcl/functions/regsuball/) — Replaces all occurrences of `pattern`, which may be a Perl-compatible regular expression, in `input` with `replacement`.

- setcookie.get_value_by_name() (/vcl/functions/setcookie-get-value-by-name/) — Returns a value associated with the `cookie_name` in the `Set-Cookie` header contained in the HTTP response indicated by `where`.

- std.atoi() (/vcl/functions/std-atoi/) — Takes a string (which represents an integer) as an argument and returns its value.

- std.ip() (/vcl/functions/std-ip/) — An alias of `std.str2ip`.

- std.ip2str() (/vcl/functions/std-ip2str/) — Converts the IP address (v4 or v6) to a string.

- std.str2ip() (/vcl/functions/std-str2ip/) — Converts the string address to an IP address (v4 or v6).

- std.strlen() (/vcl/functions/std-strlen/) — Returns the length of the string.

- std.strstr() (/vcl/functions/std-strstr/) — Finds the first occurrence of a byte string and returns its value.

- std.strtol() (/vcl/functions/std-strtol/) — Converts a string to an integer, using the second argument as base.

- std.tolower() (/vcl/functions/std-tolower/) — Changes the case of a string to lower case.

- std.toupper() (/vcl/functions/std-toupper/) — Changes the case of a string to upper case.

- subfield() (/vcl/functions/subfield/) — Provides a means to access subfields from a header like `Cache-Control`, `Cookie`, and `Edge-Control` or individual parameters from the query string.

- suffixof() (/vcl/functions/suffixof/) — True if the string `needle` ends the string `haystack`.

- urldecode() (/vcl/functions/urldecode/) — Decodes a percent-encoded string.

- urlencode() (/vcl/functions/urlencode/) — Encodes a string for use in a URL.

Query string manipulation (/vcl/query-string-manipulation/)

Fastly provides a number of extensions to VCL (/guides/vcl-tutorials/guide-to-vcl#fastlys-vcl-extensions), including several functions for query-string manipulation based on Dridi Boukelmoune's vmod-querystring (https://github.com/Dridi/libvmod-querystring) for Varnish.

- boltsort.sort() (/vcl/functions/boltsort-sort/) — Sorts URL parameters.

- querystring.add() (/vcl/functions/querystring-add/) — Returns the given URL with the given parameter name and value appended to the end of the query string.

- querystring.clean() (/vcl/functions/querystring-clean/) — Returns the given URL without empty parameters.

- querystring.filter_except() (/vcl/functions/querystring-filter-except/) — Returns the given URL but only keeps the listed parameters.

- querystring.filter() (/vcl/functions/querystring-filter/) — Returns the given URL without the listed parameters.

- querystring.filtersep() (/vcl/functions/querystring-filtersep/) — Returns the separator needed by the `querystring.filter()` and `querystring.filter_except()` functions.

- querystring.globfilter_except() (/vcl/functions/querystring-globfilter-except/) — Returns the given URL but only keeps the parameters matching a glob.

- querystring.globfilter() (/vcl/functions/querystring-globfilter/) — Returns the given URL without the parameters matching a glob.

- querystring.regfilter_except() (/vcl/functions/querystring-regfilter-except/) — Returns the given URL but only keeps the parameters matching a regular expression.

- querystring.regfilter() (/vcl/functions/querystring-regfilter/) — Returns the given URL without the parameters matching a regular expression.

- querystring.remove() (/vcl/functions/querystring-remove/) — Returns the given URL with its query-string removed.

- querystring.set() (/vcl/functions/querystring-set/) — Returns the given URL with the given parameter name set to the given value, replacing the original value and removing any duplicates.

- querystring.sort() (/vcl/functions/querystring-sort/) — Returns the given URL with its query-string sorted.

Randomness (/vcl/randomness/)

Fastly exposes a number of functions that support the insertion of random strings, content cookies, and decisions into requests.

- randombool_seeded() (/vcl/functions/randombool-seeded/) — Identical to randombool, except takes an additional parameter, which is used to seed the random number generator.

- randombool() (/vcl/functions/randombool/) — Returns a random, boolean value.

- randomint_seeded() (/vcl/functions/randomint-seeded/) — Identical to randomint, except takes an additional parameter used to seed the random number generator.

- randomint() (/vcl/functions/randomint/) — Returns a random integer value between `from` and `to`, inclusive.

- randomstr() (/vcl/functions/randomstr/) — Returns a random string of length `len` containing characters from the supplied string `characters`.

TLS and HTTP/2 (/vcl/tls-and-http2/)

Fastly has added several variables that expose information about the TLS and HTTP/2 attributes of a request.

- h2.push() (/vcl/functions/h2-push/) — Triggers an HTTP/2 server push of the asset passed into the function as the input-string.

UUID (/vcl/uuid/)

The universally unique identifier (UUID) module provides interfaces for generating and validating unique identifiers as defined by RFC4122 (https://tools.ietf.org/html/rfc4122). Version 1 identifiers, based on current time and host identity, are currently not supported.

- uuid.dns() (/vcl/functions/uuid-dns/) — Returns the RFC4122 identifier of DNS namespace, namely the constant `"6ba7b810-9dad-11d1-80b4-00c04fd430c8"`.

- uuid.is_valid() (/vcl/functions/uuid-is-valid/) — Returns true if the string holds a textual representation of a valid UUID (per RFC4122).

- uuid.is_version3() (/vcl/functions/uuid-is-version3/) — Returns true if string holds a textual representation of a valid version 3 UUID.

- uuid.is_version4() (/vcl/functions/uuid-is-version4/) — Returns true if string holds a textual representation of a valid version 4 UUID.

- uuid.is_version5() (/vcl/functions/uuid-is-version5/) — Returns true if string holds a textual representation of a valid version 5 UUID.

- uuid.oid() (/vcl/functions/uuid-oid/) — Returns the RFC4122 identifier of ISO OID namespace, namely the constant `"6ba7b812-9dad-11d1-80b4-00c04fd430c8"`.

- uuid.url() (/vcl/functions/uuid-url/) — Returns the RFC4122 identifier of URL namespace, namely the constant `"6ba7b811-9dad-11d1-80b4-00c04fd430c8"`.

- uuid.version3() (/vcl/functions/uuid-version3/) — Derives a UUID corresponding to `name` within the given `namespace` using MD5 hash function.

- uuid.version4() (/vcl/functions/uuid-version4/) — Returns a UUID based on random number generator output.

- uuid.version5() (/vcl/functions/uuid-version5/) — Derives a UUID corresponding to `name` within the given `namespace` using SHA1 hash function.

- uuid.x500() (/vcl/functions/uuid-x500/) — Returns the RFC4122 identifier of X.500 namespace, namely the constant `"6ba7b812-9dad-11d1-80b4-00c04fd430c8"`.

📄 Variables (/vcl/variables/)

These VCL variables are supported by Fastly.

Date and time (/vcl/date-and-time/)

By default VCL includes the `now` variable, which provides the current time (for example, `Mon, 02 Jan 2006 22:04:05 GMT`). Fastly adds several new Varnish variables and functions that allow more flexibility when dealing with dates and times.

- now.sec (/vcl/variables/now-sec/) — Like the `now` variable, but in seconds since the UNIX Epoch.

- now (/vcl/variables/now/) — The current time in RFC 1123 format format.

- time.elapsed.msec_frac (/vcl/variables/time-elapsed-msec-frac/) — The time the request started in milliseconds since the last whole second.

- time.elapsed.msec (/vcl/variables/time-elapsed-msec/) — The time since the request start in milliseconds.

- time.elapsed.sec (/vcl/variables/time-elapsed-sec/) — The time since the request start in seconds.

- time.elapsed.usec_frac (/vcl/variables/time-elapsed-usec-frac/) — The time the request started in microseconds since the last whole second.

- time.elapsed.usec (/vcl/variables/time-elapsed-usec/) — The time since the request start in microseconds.

- time.elapsed (/vcl/variables/time-elapsed/) — The time since the request start, using RFC 1123 format.

- time.end.msec_frac (/vcl/variables/time-end-msec-frac/) — The time the request started in milliseconds since the last whole second.

- time.end.msec (/vcl/variables/time-end-msec/) — The time the request ended in milliseconds since the UNIX Epoch.

- time.end.sec (/vcl/variables/time-end-sec/) — The time the request ended in seconds since the UNIX Epoch.

- time.end.usec_frac (/vcl/variables/time-end-usec-frac/) — The time the request started in microseconds since the last whole second.

- time.end.usec (/vcl/variables/time-end-usec/) — The time the request ended in microseconds since the UNIX Epoch.

- time.end (/vcl/variables/time-end/) — The time the request ended, using RFC 1123 format.

- time.start.msec_frac (/vcl/variables/time-start-msec-frac/) — The time the request started in milliseconds since the last whole second, after TLS termination.

- time.start.msec (/vcl/variables/time-start-msec/) — The time the request started in milliseconds since the UNIX Epoch, after TLS termination.

- time.start.sec (/vcl/variables/time-start-sec/) — The time the request started in seconds since the UNIX Epoch, after TLS termination.

- time.start.usec_frac (/vcl/variables/time-start-usec-frac/) — The time the request started in microseconds since the last whole second, after TLS termination.

- time.start.usec (/vcl/variables/time-start-usec/) — The time the request started in microseconds since the UNIX Epoch, after TLS termination.

- time.start (/vcl/variables/time-start/) — The time the request started, after TLS termination, using RFC 1123 format.

- time.to_first_byte (/vcl/variables/time-to-first-byte/) — The time interval since the request started up to the point before the `vcl_deliver` function ran.

Edge Side Includes (ESI) (/vcl/esi/)

Fastly exposes tools to allow you to track a request that has ESI.

- req.esi (/vcl/variables/req-esi/) — Whether or not to enable ESI processing during this request.

- req.topurl (/vcl/variables/req-topurl/) — In an ESI subrequest, returns the URL of the top-level request.

Geolocation (/vcl/geolocation/)

Fastly exposes a number of geographic variables for you to take advantage of inside VCL for both IPv4 and IPv6 client IPs.

- client.as.name (/vcl/variables/client-as-name/) — The name of the organization associated with `client.as.number`.

- client.as.number (/vcl/variables/client-as-number/) — The autonomous system (AS) number associated with this IP address.

- client.geo.area_code (/vcl/variables/client-geo-area-code/) — The telephone area code associated with the IP address.

- client.geo.city.ascii (/vcl/variables/client-geo-city-ascii/) — An alias of `client.geo.city`.

- client.geo.city.utf8 (/vcl/variables/client-geo-city-utf8/) — The city or town name associated with the IP address, encoded using the UTF-8 character encoding.

- client.geo.city (/vcl/variables/client-geo-city/) — The city or town name associated with the IP address, encoded using the ASCII character encoding (a lowercase ASCII approximation of the original string with diacritics removed).

- client.geo.conn_speed (/vcl/variables/client-geo-conn-speed/) — The type of connection speed associated with the IP address.

- client.geo.continent_code (/vcl/variables/client-geo-continent-code/) — A two-character code representing the continent associated with the IP address.

- client.geo.country_code (/vcl/variables/client-geo-country-code/) — A two-character ISO 3166-1 country code for the country associated with the IP address.

- client.geo.country_code3 (/vcl/variables/client-geo-country-code3/) — A three-character ISO 3166-1 alpha-3 country code for the country associated with the IP address.

- client.geo.country_name.ascii (/vcl/variables/client-geo-country-name-ascii/) — An alias of `client.geo.country_name`.

- client.geo.country_name.utf8 (/vcl/variables/client-geo-country-name-utf8/) — The country name associated with the IP address, encoded using the UTF-8 character encoding.

- client.geo.country_name (/vcl/variables/client-geo-country-name/) — The country name associated with the IP address, encoded using the ASCII character encoding (a lowercase ASCII approximation of the original string with diacritics removed).

- client.geo.gmt_offset (/vcl/variables/client-geo-gmt-offset/) — The time zone offset from coordinated universal time (UTC) for the `client.geo.city` associated with the IP address.

- client.geo.latitude (/vcl/variables/client-geo-latitude/) — The latitude associated with the IP address.

- client.geo.longitude (/vcl/variables/client-geo-longitude/) — The longitude associated with the IP address.

- client.geo.metro_code (/vcl/variables/client-geo-metro-code/) — The metro code associated with the IP address.

- client.geo.postal_code (/vcl/variables/client-geo-postal-code/) — The postal code associated with the IP address.

- client.geo.region (/vcl/variables/client-geo-region/) — The ISO 3166-2 region code associated with the IP address.

- server.datacenter (/vcl/variables/server-datacenter/) — A code representing one of Fastly's POP locations.

- server.region (/vcl/variables/server-region/) — A code representing the general region of the world in which the POP location resides.

Miscellaneous (/vcl/miscellaneous/)

Fastly has added several miscellaneous features to Varnish that don't easily fit into specific categories.

- bereq.url.basename (/vcl/variables/bereq-url-basename/) — Same as `req.url.basename`, except for use between Fastly and your origin servers.

- bereq.url.dirname (/vcl/variables/bereq-url-dirname/) — Same as `req.url.dirname`, except for use between Fastly and your origin servers.

- bereq.url.qs (/vcl/variables/bereq-url-qs/) — The query string portion of `bereq.url`.

- bereq.url (/vcl/variables/bereq-url/) — The URL sent to the backend.

- beresp.backend.ip (/vcl/variables/beresp-backend-ip/) — The IP of the backend this response was fetched from (backported from Varnish 3).

- **beresp.backend.name (/vcl/variables/beresp-backend-name/)** — The name of the backend this response was fetched from (backported from Varnish 3).

- **beresp.backend.port (/vcl/variables/beresp-backend-port/)** — The port of the backend this response was fetched from (backported from Varnish 3).

- **beresp.grace (/vcl/variables/beresp-grace/)** — Defines how long an object can remain overdue and still have Varnish consider it for grace mode.

- **beresp.hipaa (/vcl/variables/beresp-hipaa/)** — Specifies that content not be cached in non-volatile memory to help customers meet HIPAA security requirements.

- **beresp.pci (/vcl/variables/beresp-pci/)** — Specifies that content be cached in a manner that satisfies PCI DSS requirements.

- **client.port (/vcl/variables/client-port/)** — Returns the remote client port.

- **client.requests (/vcl/variables/client-requests/)** — Tracks the number of requests received by Varnish over a persistent connection.

- **client.socket.pace (/vcl/variables/client-socket-pace/)** — Ceiling rate in bytes per second for bytes sent to the client.

- **req.grace (/vcl/variables/req-grace/)** — Defines how long an object can remain overdue and still have Varnish consider it for grace mode.

- **req.http.host (/vcl/variables/req-http-host/)** — The full host name, without the path or query parameters.

- **req.is_ipv6 (/vcl/variables/req-is-ipv6/)** — Indicates whether the request was made using IPv6 or not.

- **req.restarts (/vcl/variables/req-restarts/)** — Counts the number of times the VCL has been restarted.

- **req.url.basename (/vcl/variables/req-url-basename/)** — The file name specified in a URL.

- **req.url.dirname (/vcl/variables/req-url-dirname/)** — The directories specified in a URL.

- **req.url.ext (/vcl/variables/req-url-ext/)** — The file extension specified in a URL.

- **req.url.path (/vcl/variables/req-url-path/)** — The full path, without any query parameters.

- **req.url.qs (/vcl/variables/req-url-qs/)** — The query string portion of `req.url`.

- **req.url (/vcl/variables/req-url/)** — The full path, including query parameters.

- **stale.exists (/vcl/variables/stale-exists/)** — Specifies if a given object has stale content in cache.

Size (/vcl/size/)

To allow better reporting, Fastly has added several variables to VCL to give more insight into what happened in a request.

- req.body_bytes_read (/vcl/variables/req-body-bytes-read/) — How big the body of a request was in total bytes.

- req.bytes_read (/vcl/variables/req-bytes-read/) — How big a request was in total bytes.

- req.header_bytes_read (/vcl/variables/req-header-bytes-read/) — How big the header of a request was in total bytes.

- resp.body_bytes_written (/vcl/variables/resp-body-bytes-written/) — How many bytes were written for body of a response.

- resp.bytes_written (/vcl/variables/resp-bytes-written/) — How many bytes in total were sent as a response.

- resp.completed (/vcl/variables/resp-completed/) — Whether the response completed successfully or not.

- resp.header_bytes_written (/vcl/variables/resp-header-bytes-written/) — How many bytes were written for the header of a response.

TLS and HTTP/2 (/vcl/tls-and-http2/)

Fastly has added several variables that expose information about the TLS and HTTP/2 attributes of a request.

- fastly_info.h2.is_push (/vcl/variables/fastly-info-h2-is-push/) — Whether or not this request was a server-initiated request generated to create an HTTP/2 Server-pushed response.

- fastly_info.h2.stream_id (/vcl/variables/fastly-info-h2-stream-id/) — If the request was made over HTTP/2, the underlying HTTP/2 stream ID.

- fastly_info.is_h2 (/vcl/variables/fastly-info-is-h2/) — Whether or not the request was made using http2.

- tls.client.cipher (/vcl/variables/tls-client-cipher/) — The cipher suite used to secure the client TLS connection.

- tls.client.ciphers_sha (/vcl/variables/tls-client-ciphers-sha/) — A SHA1 of the cipher suite identifiers sent from the client as part of the TLS handshake, represented in base64.

- tls.client.protocol (/vcl/variables/tls-client-protocol/) — The TLS protocol version this connection is speaking over.

- tls.client.servername (/vcl/variables/tls-client-servername/) — The Server Name Indication (SNI) the client sent in the `ClientHello` TLS record.

- tls.client.tlsexts_sha (/vcl/variables/tls-client-tlsexts-sha/) — A SHA1 of the TLS extension identifiers sent from the client as part of the TLS handshake, represented in base64.

📄 Local variables (/vcl/local-variables/)

Fastly VCL (/guides/vcl/guide-to-vcl) supports variables for storing temporary values during request processing.

> ★ **TIP:** Consider using a `req.http.*` header to store a value if you need to pass information between functions or to the origin.

Declaring a variable

Variables must be declared before they are used, usually at the beginning of a function before any statements. They can only be used in the same function where they are declared. Fastly VCL does not provide block scope. Declarations apply to an entire function's scope even if a variable is declared within a block.

Variables start with `var.` and their names consist of characters in the set `[A-Za-z0-9._-]`. (`:` is explicitly disallowed.) The declaration syntax is:

```
declare local var.<name> <type>;
```

Variable types

Variables can be of the following types:

- `BOOL` (/vcl/types/bool/)

- `FLOAT` (/vcl/types/float/)

- `INTEGER` (/vcl/types/integer/)

- `IP` (/vcl/types/ip/)

- `RTIME` (/vcl/types/rtime/) (relative time)

- `STRING` (/vcl/types/string/)

- `TIME` (/vcl/types/time/) (absolute time)

Declared variables are initialized to the zero value of the type:

- `0` for numeric types

- `false` for `BOOL`

- `NULL` for `STRING`

Usage

Boolean variables

Boolean assignments support boolean variables on the right-hand side as well as `BOOL`-returning functions, conditional expressions, and the `true` and `false` constants.

```
1    declare local var.boolean BOOL;
2
3    # BOOL assignment with RHS variable
4    set var.boolean = true;
5    set req.esi = var.boolean;
6    set resp.http.Bool = if(req.esi, "y", "n");
7
8    # BOOL assignment with RHS function
9    set var.boolean = http_status_matches(resp.status, "200,304");
10
11   # BOOL assigment with RHS conditional
12   set var.boolean = (req.url == "/");
13
14   # non-NULL-ness check, like 'if (req.http.Foo) { ... }'
15   set var.boolean = (req.http.Foo);
```

Numeric variables

Numeric assignment and comparison support numeric variables (anything except `STRING` or `BOOL`) on the right-hand side, including conversion in both directions between `FLOAT` and `INTEGER` types, rounding to the nearest integer in the `FLOAT` to `INTEGER` case.

Invalid conditions or domain errors like division by 0 will set `fastly.error`.

```
1    declare local var.integer INTEGER;
2    declare local var.float FLOAT;
3
4    # Numeric assignment with RHS variable and
5    # implicit string conversion for header
6    set var.integer = req.bytes_read;
7    set var.integer -= req.body_bytes_read;
8    set resp.http.VarInteger = var.integer;
9
10   # Numeric comparison with RHS variable
11   set resp.http.VarIntegerOK = if(req.header_bytes_read == var.integer, "y", "n");
```

String variables

String assignments support string concatenation on the right-hand side.

```
1   declare local var.restarted STRING;
2
3   # String concatenation on RHS
4   set var.restarted = "Request " if(req.restarts > 0, "has", "has not") " restarted."
    ;
```

IP address variables

IP address variables represent individual IP addresses.

```
1   acl office_ip_ranges {
2     "192.0.2.0"/24;                          # internal office
3     "198.51.100.4";                          # remote VPN office
4     "2001:db8:ffff:ffff:ffff:ffff:ffff:ffff";    # ipv6 address remote
5   }
6
7   declare local var.ip1 IP;
8   set var.ip1 = "192.0.2.0";
9
10  if (var.ip1 ~ office_ip_ranges) {
11    ...
12  }
13
14  declare local var.ip2 IP;
15  set var.ip2 = "2001:db8:ffff:ffff:ffff:ffff:ffff:ffff";
```

Time variables

Time variables support both relative and absolute times.

```
1   declare local var.time TIME;
2   declare local var.rtime RTIME;
3
4   set req.grace = 72s;
5   set var.rtime = req.grace;
6   set resp.http.VarRTime = var.rtime;
7
8   set var.time = std.time("Fri, 10 Jun 2016 00:02:12 GMT", now);
9   set var.time -= var.rtime;
10  # implicit string conversion for header
11  set resp.http.VarTime = var.time;
```

📄Operators (/vcl/operators/)

Fastly's VCL provides various arithmetic and conditional operators. Operators are syntactic items which evaluate to a value. Syntax is given in a BNF-like form with the following conventions:

- […] Square brackets enclose an optional item,

- "!" Literal spellings (typically punctuation) are indicated in quotes,

- CNUM Lexical terminals are given in uppercase,

- INTEGER Types are also given in uppercase,

- numeric-expr Grammatical productions are given in lowercase.

Where a binary operator is provided, not all types are implemented on either side. This is a limitation of the current implementation. The following placeholder grammatical clauses are used in this document to indicate which types are valid operands. These are not precisely defined until the grammar has been formally specified, and are intended as a guide for operator context only.

- variable - A variable name

- acl - An ACL name

- expr - An expression of any type

- numeric-expr - An expression evaluating to INTEGER, FLOAT, RTIME, or another numeric type

- time-expr - An expression evaluating to TIME

- assignment-expr - An expression suitable for assignment to a variable by `set`

- conditional-expr - An expression evaluating to BOOL suitable for use with `if` conditions

- string-expr - An expression evaluating to STRING

- CNUM - An INTEGER literal

Operator precedence

Operator precedence defines the *order of operations* when evaluating an expression. Higher precedence operators are evaluated before those with lower precedence. Operators are listed in the following table as the highest precedence first. For example, `a || b && c` reads as `a || (b && c)` because `&&` has higher precedence than `||`.

Operator *associativity* determines which side binds first for multiple instances of the same operator at equal precedence. For example, `a && b && c` reads as `(a && b) && c` because `&&` has left to right associativity.

Operator	Name	Associativity
`(` `)`	Grouping for precedence	left to right
`!`	Boolean NOT	right to left
`&&`	Boolean AND	left to right

Operator	Name	Associativity
`||`	Boolean OR	left to right

Negation

Numeric literals may be negated by prefixing the `-` unary operator. This operator may only be applied to literals, and not to numeric values in other contexts.

```
1    := [ "−" ] CNUM
2     | [ "−" ] CNUM "." [ CNUM ]
```

String concatenation

Adjacent strings are concatenated implicitly, but may also be concatenated explicitly by the `+` operator:

```
1    := string−expr string−expr
2     | string−expr "+" _string−expr
```

For example, `"abc" "def"` is equivalent to `"abcdef"`.

Assignment and arithmetic operators

The `set` syntax is the only situation in which these operators may be used. Since the operator may only occur once in a `set` statement, these operators are mutually exclusive, so precedence between them is nonsensical.

The values the operators produce are used for assignment only. The `set` statement assigns this value to a variable, but does not itself evaluate to a value.

FLOAT arithmetic has special cases for operands which are NaN: Arithmetic operators evaluate to NaN when either operand is NaN.

FLOAT arithmetic has special cases for operands which are floating point infinities: In general all arithmetic operations evaluate to positive or negative infinity when either operand is infinity. However some situations evaluate to NaN instead. Some of these situations are *domain errors*, in which case `fastly.error` is set to `"EDOM"` accordingly. Others situations are not domain errors: ∞ − ∞ and 0 × ∞. These evaluate to NaN but do not set `fastly.error`.

Assignment

Assignment is provided by the `=` operator:

```
1    := "set" variable "=" assignment−expr ";"
```

Addition and subtraction

Addition and subtraction are provided by the `+=` and `-=` operators respectively:

```
1    := "set" variable "+=" assignment-expr ";"
2     | "set" variable "-=" assignment-expr ";"
```

Multiplication, division and modulus

Multiplication, division and modulus are provided by the `*=`, `/=` and `%=` operators respectively:

```
1    := "set" variable "*=" assignment-expr ";"
2     | "set" variable "/=" assignment-expr ";"
3     | "set" variable "%=" assignment-expr ";"
```

Bitwise operators

```
1    := "set" variable "|=" assignment-expr ";"
2     | "set" variable "&=" assignment-expr ";"
3     | "set" variable "^=" assignment-expr ";"
4     | "set" variable ">>=" assignment-expr ";"
5     | "set" variable "<<=" assignment-expr ";"
6     | "set" variable "ror=" assignment-expr ";"
7     | "set" variable "rol=" assignment-expr ";"
```

Right shifts sign-extend negative numbers. For example, `-32 >> 5` gives -1.

Shift and rotate operations with negative shift widths perform the operation in the opposite direction. For example, `32 << -5` gives 1. For right operands larger than the width of `INTEGER`, shifts will yield zero or -1 and rotates will use the operand modulo the width of `INTEGER`.

Logical operators

Logical AND and OR operators are provided by the `&&=` and `||=` operators respectively:

```
1    := "set" variable "&&=" assignment-expr ";"
2     | "set" variable "||=" assignment-expr ";"
```

These are *short-circuit* operators; see below.

Conditional operators

Conditional operators produce BOOL values, suitable for use in `if` statement conditions.

Logical operators

Conditional expressions may be inverted by prefixing the `!` operator:

```
1    := "!" conditional-expr
```

Boolean AND and OR operators (`&&` and `||` respectively) are defined for conditional expressions:

```
1    := conditional-expr "&&" conditional-expr
2     | conditional-expr "||" conditional-expr
```

These boolean operators have *short-circuit* evaluation, whereby the right-hand operand is only evaluated when necessary in order to compute the resulting value. For example, given `a && b` when the left-hand operand is false, the resulting value will always be false, regardless of the value of the right-hand operand. So in this situation, the right-hand operand will not be evaluated. This can be seen when the right-hand operand has a visible side effect, such as a call to a function which performs some action.

Comparison operators

FLOAT comparisons have special cases for operands which are NaN: The `!=` operator always evaluates to true when either operand is NaN. All other conditional operators always evaluate to false when either operand is NaN. For example, if a given variable is NaN, that variable will compare unequal to itself: both `var.nan == var.nan` and `var.nan >= var.nan` will be false.

STRING comparisons have special cases for operands which are not set (as opposed to empty): The `!=` and `!~` operators always evaluate to true when either operand is not set. All other conditional operators always evaluate to false when either operand is not set. For example, if a given variable is not set, that variable will compare unequal to itself: both `req.http.unset == req.http.unset` and `req.http.unset ~ ".?"` will be false.

Floating point infinities are signed, and compare as beyond the maximum and minimum values for FLOAT types, such that for any finite value: $-\infty < n < +\infty$

The comparison operators are:

```
1    lg-op := "<" | ">" | "<=" | ">="
2    eq-op := "==" | "!="
3    re-op := "~" | "!~"
```

Equality is defined for all types:

```
1    := expr eq-op expr
```

Inequalities are defined for numeric types and TIME:

```
1    := numeric-expr lg-op numeric-expr
2     | time-expr lg-op time-expr
```

Note that as there are currently no numeric expressions in general; these operators are limited to use with specific operands. For example, `var.i < 5` is permitted but `2 < 5` is not.

Regular expression conditional operators are defined for STRING types and ACLs only:

```
1   := string-expr re-op STRING
2    | acl re-op STRING
```

The right-hand operand must be a literal string (regular expressions cannot be constructed dynamically).

Reserved punctuation

Punctuation appears in various syntactic roles which are not operators (that is, they do not produce a value).

Punctuation	Example Uses
`{` `}`	Block syntax
`[` `]`	Stats ranges
`(` `)`	Syntax around if conditions, function argument lists
`/`	Netmasks for ACLs
`,`	Separator for function arguments
`;`	Separator for statements and various other syntactic things
`!`	Invert ACL entry
`.`	To prefix fields in backend declarations
`:`	Port numbers for backend declarations, and used in the stats syntax

The following lexical tokens are reserved, but not used: `*` `&` `|` `>>` `<<` `++` `--` `%`

📄 Types (/vcl/types/)

VCL is a statically typed language. Several types are available.

Types for scalar values

These types are provided for scalar values, and may be assigned values from literals. Some types have units; others are unitless.

These types all have implicit conversions to strings, such that their values may be used in contexts where a STRING value is necessary. The rendering for string conversion is not described except for types where it differs from the corresponding literal syntax.

- BOOL (/vcl/types/bool/)

- FLOAT (/vcl/types/float/)

- INTEGER (/vcl/types/integer/)

- IP (/vcl/types/ip/)

- RTIME (/vcl/types/rtime/)

- STRING (/vcl/types/string/)

- TIME (/vcl/types/time/)

Types with special semantics

These types serve as points of abstraction, where internal mechanisms are separated from their interfaces to the VCL syntax. This is either due to special cases for syntax in VCL, or provided for special cases for operations internally.

- BACKEND (/vcl/types/backend/)

- HASH (/vcl/types/hash/)

- HEADER (/vcl/types/header/)

- VOID (/vcl/types/void/)