

# Parallels<sup>®</sup> Plesk Expand

---

Developer Guide

## Expand API

Plesk Expand 2.3

# Contents

<b>Preface</b>	<b>3</b>
About This Guide .....	3
Who Should Read This Guide .....	3
Typographical Conventions .....	4
Feedback .....	4
<b>Expand API Overview</b>	<b>5</b>
<b>Forming Request Messages</b>	<b>7</b>
Forming Request Packets .....	8
Packet Syntax .....	8
Packet Samples .....	9
How to Create Request Packet .....	11
Forming Request Statements .....	12
<b>Invoking Expand Operations</b>	<b>14</b>
Using External Endpoint to Invoke Operations .....	14
Using CLI to Invoke Operations .....	16
<b>Processing Response Packets</b>	<b>17</b>
<b>Client Application Samples</b>	<b>20</b>
PHP Client .....	20
Perl Client .....	23

# Preface

## In this section:

About This Guide.....	3
Who Should Read This Guide .....	3
Typographical Conventions .....	4
Feedback .....	4

---

## About This Guide

This part of Expand API documentation familiarizes you with Expand API and explains how to use it.

This guide is organized as follows:

- Chapter **Expand API Overview** (on page 5) briefly acquaints you with Expand API, its architecture and resources.
- Chapter **Forming Request Messages** (on page 8) introduces Expand API Protocol and provides the basis of forming protocol messages. These messages describe Expand resources and operations assigned to them.
- Chapter **Invoking Expand Operations** (on page 14) discusses how to invoke an operation using a given request message.
- Chapter **Processing Response Packets** (on page 17) describes what data is returned by Expand after it attempts to perform an operation. This chapter also provides instructions on how to handle this data.
- Chapter **Client Application Samples** (on page 20) presents sample client applications that illustrate how to utilize the Expand API.

---

## Who Should Read This Guide

This guide is intended for developers who want to integrate Expand with third-party software or automate Expand business processes.

---

# Typographical Conventions

Before you start using this guide, it is important to understand the documentation conventions used in it.

The following kinds of formatting in the text identify special information.

Formatting convention	Type of Information	Example
<b>Special Bold</b>	Items you must select, such as menu options, command buttons, or items in a list.	Go to the <b>System</b> tab.
	Titles of chapters, sections, and subsections.	Read the <b>Basic Administration</b> chapter.
<i>Italics</i>	Used to emphasize the importance of a point, to introduce a term or to designate a command line placeholder, which is to be replaced with a real name or value.	The system supports the so called <i>wildcard character</i> search.
Monospace	The names of commands, files, and directories.	The license file is located in the <code>http://docs/common/licenses</code> directory.
Preformatted	On-screen computer output in your command-line sessions; source code in XML, C++, or other programming languages.	<pre># ls -al /files total 14470</pre>
<b>Preformatted Bold</b>	What you type, contrasted with on-screen computer output.	<pre># cd /root/rpms/php</pre>
CAPITALS	Names of keys on the keyboard.	SHIFT, CTRL, ALT
KEY+KEY	Key combinations for which the user must press and hold down one key and then press another.	CTRL+P, ALT+F4

---

## Feedback

If you have found a mistake in this guide, or if you have suggestions or ideas on how to improve this guide, please send your feedback using the online form at <http://www.parallels.com/en/support/usersdoc/>. Please include in your report the guide's title, chapter and section titles, and the fragment of text in which you have found an error.

# Expand API Overview

Expand provides API to perform operations on its resources. These resources include Expand, Plesk, and Virtuozzo manageable objects. In Expand API, each such object is mapped to a specific XML schema that identifies properties and operations assigned to the object. To perform an operation on a specific object, an application that utilizes the API must create an XML message formed according to a certain XML schema, and send it to Expand. The latter receives the XML message, performs the operation, and responds with an XML message containing the operation details. Thus, Expand API is a client-server interface: The server part (hereinafter called *server*) is the Expand server; the client part (hereinafter called *client application* or simply *client*) is a third-party application that access Expand resources via the API.

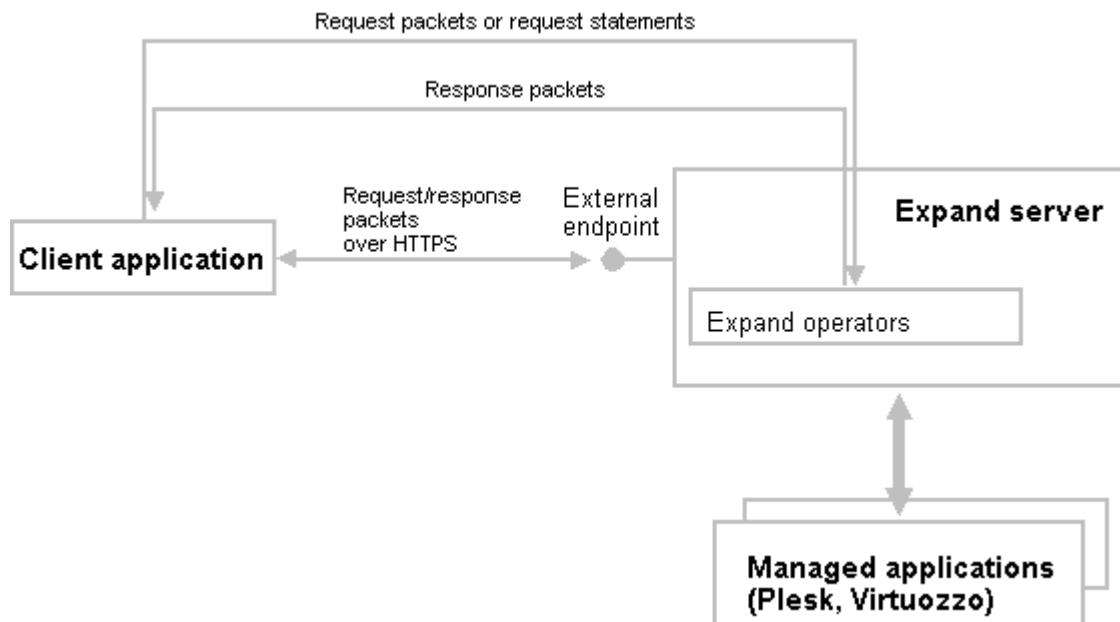


Figure 1: Expand API

As mentioned, interaction between the server and clients implies exchanging XML messages. We will refer to each such message as to a *packet*. A packet issued by a client we will call *request packet*. A packet issued by the server we will call *response packet*.

Packets are transported between clients and the server in two different ways:

- Remotely via HTTPS. A client wraps a request packet into an HTTP request message-body and sends it to the Expand external endpoint. In this case, the response packet is wrapped into the HTTP response message-body.
- Locally via CLI. Expand provides a set of command line utilities called *operators* that process request packets. If a client has access to operators, it can perform a specific operation by running an appropriate operator. Operation parameters defined in a packet are read by the operator from `STDIN`. The response packet is sent to `STDOUT`.

---

**Note:** To prevent unauthorized access to Expand API, a client must be authorized as Expand Administrator. If a client uses CLI, Administrator's credentials are not verified (it is supposed that only Expand Administrator has access to operators). Otherwise, a client must provide Administrator's credentials in a HTTP header.

---

If a client uses CLI, it can alternately invoke operations by running operators with command line options. These options are called *request statements* - data in specific format that describes an operation to be performed. In this case, the response packet is sent to `STDOUT` too.

Both request statements and request packets are called *request messages*. A single request message can be processed only by a particular operator. The syntax and semantics of request messages and response packets are defined by the Expand API Protocol hereafter called simply *Protocol*. For details on the Protocol, refer to the **Expand API Protocol Reference** section of the Expand API Reference document. For details on how to create a valid request message, refer to the next chapter.

# Forming Request Messages

Clients interact with the server by issuing request messages and retrieving response packets. This chapter provides instructions on how to form different types of request messages (request packets and request statements). The instructions are accompanied by samples.

## In this chapter:

Forming Request Packets .....	8
Forming Request Statements .....	12

# Forming Request Packets

This section familiarizes you with the syntax of the packets, provides packet samples, and explains how to create a request packet.

## In this section:

Packet Syntax .....	8
Packet Samples .....	9
How to Create Request Packet .....	11

## Packet Syntax

The syntax of a packet is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<packet version="version">
  <operation_1>
    <parameter_1>...</parameter_1>
    [...]
    <parameter_k>...</parameter_k>]
  </operation_1>
  [...]
  <operation_n>
    <parameter_1>...</parameter_1>
    [...]
    <parameter_m>...</parameter_m>]
  </operation_n>]
</packet>
```

The *[operation]* element defines the operation performed on a specific object which is defined by the *[parameter]* element along with other operation-related data. Practically, there can be multiple operations within a single `packet` element.

To form an appropriate packet, you need to substitute *operation* and *parameter* placeholders with the values conforming to the Protocol. This protocol also defines semantic information related to each operation. For details on the syntax and semantics of a specific operation, refer to **Expand API Protocol Reference > Objects and Operations** section of the Expand API Reference document.

## Packet Samples

This section presents samples of request and response packets.

The following request packet is used for creating a Plesk domain account. The packet is divided into the *packet header* and *packet body* for convenience of description.

<code>&lt;?xml version="1.0" encoding="UTF-8"?&gt;</code>	<b>XML declaration</b>
<code>&lt;packet version="2.3.0.26"&gt;</code>	<b>Packet header</b> The <code>version</code> attribute specifies the version of the Protocol.
<code>&lt;add&gt;       &lt;gen_setup&gt;         &lt;name&gt;example.com&lt;/name&gt;         &lt;client_id&gt;11&lt;/client_id&gt;         &lt;status&gt;0&lt;/status&gt;         &lt;use_ip&gt;           &lt;ip_address&gt;             192.0.2.108           &lt;/ip_address&gt;         &lt;/use_ip&gt;       &lt;/gen_setup&gt;       &lt;preferences&gt;         &lt;www&gt;&gt;true&lt;/www&gt;       &lt;/preferences&gt;     &lt;/add&gt;</code>	<b>Packet body</b> The <code>add</code> element instructs the server to add a domain account.  The elements nested within the <code>&lt;gen_setup&gt;</code> and <code>&lt;preferences&gt;</code> nodes specify the domain properties.
<code>&lt;/packet&gt;</code>	<b>Trailing tag closing the packet</b>

A response packet retrieved from the server after creating a Plesk domain account can look as follows:

<code>&lt;?xml version="1.0" encoding="UTF-8" standalone="no"?&gt;</code>	<b>XML declaration</b>
<code>&lt;packet version="2.3.0.26"&gt;</code>	<b>Packet header</b>
<code>&lt;add&gt;       &lt;result&gt;         &lt;status&gt;ok&lt;/status&gt;         &lt;id&gt;20&lt;/id&gt;         &lt;client_id&gt;11&lt;/client_id&gt;         &lt;server_id&gt;2&lt;/server_id&gt;       &lt;/result&gt;     &lt;/add&gt;</code>	<b>Packet body</b> The <code>add</code> element indicates that the server attempted to add a domain account.  The <code>status</code> within this node indicates that the operation was successfully performed. In addition, the Expand server passes back the domain, Plesk client and Plesk server ID.
<code>&lt;/packet&gt;</code>	<b>Trailing tag closing the packet</b>

If the operation fails, the response packet can look as follows:

<pre>&lt;?xml version="1.0" encoding="UTF-8" standalone="no"?&gt;</pre>	<b>XML declaration</b>
<pre>&lt;packet version="2.3.0.26"&gt;</pre>	<b>Packet header</b>
<pre>&lt;add&gt;   &lt;result&gt;     &lt;status&gt;error&lt;/status&gt;     &lt;errcode&gt;4003&lt;/errcode&gt;     &lt;errtext&gt;[Operator] PleskAgent error. Error Plesk server answer: (1024) [PleskAgent - GENERAL] Reached limit. The limit on number of domains is reached for this client account.&lt;/errtext&gt;     &lt;client_id&gt;11&lt;/client_id&gt;     &lt;server_id&gt;2&lt;/server_id&gt;   &lt;/result&gt; &lt;/add&gt;</pre>	<p><b>Packet body</b></p> <p>The <code>add</code> element indicates that the server attempted to add a domain account .</p> <p>The <code>status</code> within this node indicates that the operation failed. Also, the Expand server passes back Plesk client and Plesk server ID.</p>
<pre>&lt;/packet&gt;</pre>	<b>Trailing tag closing the packet</b>

## How to Create Request Packet

The following instructions can help you create a request packet:

- 1 Specify the XML declaration and packet header, substituting the *version* placeholder with a certain Protocol version.

Each Expand version supports only a particular Protocol version. Practically, a protocol version must be the same as the Expand version so that the latter can successfully perform protocol-defined operations. There is no back compatibility between different Protocol versions.

```
<?xml version="1.0" encoding="UTF-8"?>
<packet version="version">
```

- 2 Open the Expand API Reference document. Locate the **Objects & Operations** section. In this section, choose the section that describes an operation you want to invoke, and open this section.

For instance, if you want to invoke operations on Plesk domain accounts, choose the **Managing Plesk Domain Accounts** section.

- 3 Form an operation description basing on the information provided in the section.
  - The introductory part of the section explains the operation semantics.
  - The **Request Packet Structure** sub-section describes the syntax of a request packet requesting the operation.

- 4 Add the operation description to the request packet.

- 5 Repeat steps 4, 5 and 6 for each operation.

- 6 After you specified descriptions of all operations, add the trailing tag to the end of the request packet.

```
</packet>
```

Once a request packet is ready, clients can use it to invoke operations on Expand resources, or form request statements.

## Forming Request Statements

A request statement is a string containing operation-specific data. To form a request statement, you first need to create a request packet and then transform it to the request statement. To perform this transformation, use the following guidelines:

- List all start tags corresponding to an operation in series. If the element content is not empty, put the "=" character after the element and then specify the content. Do not place white spaces before and after the "=" character. String values must be enclosed in quotes.

- 

Request packet:

```
<packet version="0.1.1.0">
  <del>
    <filter>
      <id>3</id>
    </filter>
  </del>
</packet>
```

Request statement:

```
del filter id=3
```

- If an element type is empty (`<filter/>`), append the "-" or "[]" construction to the element name. White spaces before and after brackets are important.

Request packet:

```
<packet version="0.1.1.0">
  <del>
    <filter/>
  </del>
</packet>
```

Request statement:

```
del filter []
```

- (Optional). You can use square brackets to separate children of one element from children of sibling elements. White spaces before and after brackets are important.

Request packet:

```
<add>
  <gen_setup>
    <name>Basic</name>
    <htype>vrt_hst</htype>
  </gen_setup>
  <hosting>
    <vrt_hst>
      <fp>false</fp>
      <fp_ssl>false</fp_ssl>
    </vrt_hst>
  </hosting>
</add>
```

Request statement:

```
add gen_setup [ name='First of all' htype=vrt_hst ] hosting
vrt_hst [ fp=false fp_ssl=false ]
```

Once a request statement is formed, clients can use it to invoke operations on Expand resources.

# Invoking Expand Operations

Each operation on Expand resources is performed at the server part. To invoke operations, use one of the following ways:

- Send a request packet to the Expand external endpoint
- Call a particular operator from CLI

This chapter discusses how to perform each of these actions. It also contains generic information about Expand operators and their capabilities.

## In this chapter:

Using External Endpoint to Invoke Operations .....	14
Using CLI to Invoke Operations.....	16

---

## Using External Endpoint to Invoke Operations

Client typically uses external endpoint for invoking operations when it interacts with Expand from remote. This invocation method implies that a client wraps a request packet into an HTTP POST message-body and sends it to the endpoint URL (typically, `http://<expand-host-name>:<port>/webgate.php`) via HTTPS . The client must also specify the following information in the HTTP header:

- Name of the operator which invokes requested operations (HTTP\_AUTH\_OP parameter)
- Credentials of Expand Administrator (HTTP\_AUTH\_LOGIN and HTTP\_AUTH\_PASSWORD parameters).

The credentials are verified by the server before performing operations to prevent unauthorized access to its resources. After the server attempts to perform requested operations, it returns operation details as a response packet wrapped into HTTP POST message to the client.

You can use any library that implements HTTPS to send HTTP POST messages with request packets. The following PHP script demonstrates how this can be done using the CURL engine - a free and open client-side library supported by PHP 4.0.2 and higher.

```
<?php
function write_callback($ch, $data) {
    echo $data;
    return strlen($data);
}

function sendCommand($operator_name, $data, $login, $passwd, $host,
$port=8442) {
    $script = "webgate.php";
    $url = "https://$host:$port/$script";
    $headers = array(
        "HTTP_AUTH_OP: $operator_name",
        "HTTP_AUTH_LOGIN: $login",
        "HTTP_AUTH_PASSWD: $passwd",
        "Content-Type: text/xml",
        "Expect:",
    );
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, FALSE);
    curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, FALSE);
    curl_setopt($ch, CURLOPT_HTTPHEADER, &$amp;$headers);
    curl_setopt($ch, CURLOPT_URL, $url);
    curl_setopt($ch, CURLOPT_WRITEFUNCTION, write_callback);
    curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
    curl_setopt($ch, CURLOPT_VERBOSE, 1);
    $result = curl_exec($ch);
    if (!$result) {
        echo "\n\n-----\ncURL error
number:".curl_errno($ch);
        echo "\n\ncURL error:".curl_error($ch);
    }

    curl_close($ch);
    return;
}
$data = <<<EOF
<?xml version="1.0" encoding="UTF-8"?>
<!-- Get plesk servers list -->
<packet version="2.0.1.2">
    <get>
        <filter></filter><!-- select all Plesk servers -->
        <dataset>
            <gen_info></gen_info>
        </dataset>
    </get>
</packet>
EOF;
$operator_name="exp_plesk_domain";
$login="admin";
$passwd="setup";
$host="expand_server.com";
$port=8442;

sendCommand($operator_name, $data, $login, $passwd, $host, $port);
?>
```

---

## Using CLI to Invoke Operations

Client typically uses CLI for invoking operations when it shares the same local network with the server and can execute operators.

Expand operator is an executable file that performs operations supported by a specific set of Expand resources. For instance, the `exp_plesk_client` operator performs operations on Plesk client accounts; the `exp_plesk_domain` operator performs operations on Plesk domain accounts. For details on what resources are served by a specific operator, refer to **Expand API Protocol Reference > Objects and Operations** section of the Expand API Reference .

To invoke operations, a client must run an operator with operation-specific data. The data can be of one of the following types:

- Request packet
- Request statement

---

**Note:** Operators require root privileges to run.

---

After a client runs an operator, it outputs operation details to STDOUT.

Follow the given below instructions to invoke operations using operators.

### Calling an operator with a request packet

Preconditions: You have a request packet written to a file, and you know the name of the operator that performs operations described by this packet.

To invoke the operation, run the following command in the command line:

```
# /usr/local/expand/sbin/<operator name> < <file name>
```

Example:

```
# /usr/local/expand/sbin/exp_plesk_domain < sample.xml
```

### Calling an operator with a request statement

Preconditions: You know the name of the operator that performs a desired operation, and you have an operation-specific request statement.

To invoke the operation, run the following command in the command line:

```
# /usr/local/expand/sbin/<operator name> <request statement>
```

Example:

```
# /usr/local/expand/sbin/exp_plesk_domain del filter id=3
```

# Processing Response Packets

After a request message is received, the server validates it, attempts to perform requested operations, and returns a response packet to a client. If an operation is successfully performed, the response packet contains the following operation details:

- The `result` element with the child `status` element set to `ok`.
- Operation parameters. These parameters typically include the identifier and properties of the Expand resources affected by the operation.

Otherwise, the server reports an error. The errors are divided into *check-up errors*, *service errors*, and *input errors*. All errors are identified by an error code. Error code descriptions are found in the **Expand API Protocol Reference > Error Codes** section of the Expand API Reference.

## Check-up errors

Incoming HTTP message passes through a number of check-ups on the server side before it is considered valid and ready for execution. Here are some of these check-ups:

- check-up of HTTP-POST header
- check-up of whether the Protocol version specified is supported
- check-up of the sender's credentials
- check-up of the packet contents validity

Any of these checkups can fail which makes further checkups and operation execution impossible. Since the server side has got stuck at one of the preliminary steps (i.e., it has not come to executing the requested operation yet), the response packet is formed with the `system_error` element containing an error description. In this case, operations nested in a packet are not performed.

For example, the response received from the server at the attempt to use a non-existing version of the Protocol looks as follows:

```
<packet version="2.3.0.26">
  <system_error>
    <code>1102</code>
    <text>[XML] Cannot parse packet - invalid XML or DOM error. Error
at file 'systemId', line
      27, column 16: Unknown element 'gen_setup'.</text>
  </system_error>
</packet>
```

## Service errors

If all checkups are passed through successfully, the server attempts to execute the requested operations in series. If an Expand fails to provide one of its services or some Expand resources are corrupted, the server signals this fact by adding the `system` element to the response packet and proceeds to the next operation in a packet. For instance, if the Plesk domain account data is corrupted, the server adds the following fragment to the response packet:

```
<get>
  <system>
    <status>error</status>
    <errcode>4000</errcode>
    <errtext>Tried to convert "NULL" to a "j"</errtext>
  </system>
</get>
```

## Input errors

These errors occur when a client provide incorrect data from Expand object model standpoint, or a client is not permitted to perform an operation. These errors occur, for instance, if a client tries to access an object that does not exist. The server indicates input errors by adding a `result` node to the response packet with the `status` node set to `error`.

A fragment of a response packet describing an input error is provided in the following sample:

```
<add>
  <result>
    <status>error</status>
    <errcode>4003</errcode>
    <errtext>[Operator] PleskAgent error. Error Plesk server
answer: (1024) [PleskAgent - GENERAL] Reached limit. The limit on
number of domains is reached for this client account.</errtext>
    <client_id>11</client_id>
    <server_id>2</server_id>
  </result>
</add>
```

## Processing routine

Summing up, a client should perform the following sequence of actions when it receives a response packet:

- 1 Check if the packet contains the `system` element. This element signals that a check-up error occurred.
  - If the element is present in the packet, output the error description. Check-up errors typically occur if a request packet does not conform to the Protocol.
  - If the element is not present in the packet, go to step 2.
- 2 Process the details of each operation. For details on the syntax of operation-specific data returned by the server, refer to **Supported Operations > [Operation-specific section] > Response Packet Structure**.
  - If an operation failed, output the error description and proceed to processing details of the next operation.
  - If an operation succeeded, issue the success message and proceed to processing details of the next operation.

For the code of a client application that implements the routine, refer to the next chapter.

# Client Application Samples

This chapter presents sample client applications in PHP and Perl that illustrate how to utilize the Expand API.

## In this chapter:

PHP Client .....	20
Perl Client .....	23

---

## PHP Client

The following code sample presents ready-to-use client application written in PHP. This application interacts with the Expand server via the external endpoint. The client requests information on Plesk servers managed by a certain Expand server, and processes the server response. To perform custom operations, substitute the request packet contents.

---

**Note:** If you want to use this sample, make sure you have PHP 4.0.2 or later. For more information, visit <http://php.net/manual/en/ref.dom.php> or <http://php.net/manual/en/ref.curl.php>.

---

```
<?php
function write_callback($ch, $data) {
    echo $data;
    return strlen($data);
}

function sendCommand($operator_name, $data, $login, $passwd, $host,
    $port=8442) {
    $script = "webgate.php";
    $url = "https://$host:$port/$script";
    $headers = array(
        "HTTP_AUTH_OP: $operator",
        "HTTP_AUTH_LOGIN: $login",
        "HTTP_AUTH_PASSWD: $passwd",
        "Content-Type: text/xml",
        "Expect:",
    );

    $ch = curl_init();
    curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, FALSE);
    curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, FALSE);
    curl_setopt($ch, CURLOPT_HTTPHEADER, &$headers);
    curl_setopt($ch, CURLOPT_URL, $url);
    curl_setopt($ch, CURLOPT_WRITEFUNCTION, write_callback);
    curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
```

```

    curl_setopt($ch, CURLOPT_VERBOSE, 1);
    $result = curl_exec($ch);
    if (!$result) {
        echo "\n\n-----\n\nURL error
number:".curl_errno($ch);
        echo "\n\nURL error:".curl_error($ch);
    }

    curl_close($ch);
    return;
}

$data = <<<EOF
<?xml version="1.0" encoding="UTF-8"?>
<!-- Get plesk servers list -->
<packet version="2.0.1.2">
    <get>
        <filter></filter><!-- select all Plesk servers -->
        <dataset>
            <gen info></gen info>
        </dataset>
    </get>
</packet>
EOF;
$operator_name="exp_plesk_domain";
$login="root";
$password="setup";
$host="expand_server_example.com";
$port=8442;

sendCommand($operator_name, $data, $login, $password, $host, $port);
?>

```

To tailor this application to your needs, replace the following variable values with the appropriate content:

Variable	Description
\$host	Expand host name
\$port	Expand port number (Default: 8442)
\$login	Expand Administrator login
\$password	Expand Administrator password
\$operator_name	Operator name
\$data	Request packet contents

This table explains the client application code.

Operation / group of operations	Action performed
<pre>function write_callback(\$ch, \$data) { echo \$data; return strlen(\$data); }  function sendCommand(\$operator, \$data, \$login, \$passwd, \$host, \$port=8442)  \$script = "webgate.php"; \$url = "https://\$host:\$port/\$script";  \$headers = array( "HTTP_AUTH_OP: \$operator", "HTTP_AUTH_LOGIN: \$login", "HTTP_AUTH_PASSWD: \$passwd", "Content-Type: text/xml", "Expect:", );  \$ch = curl_init(); curl_setopt(\$ch, CURLOPT_SSL_VERIFYHOST, FALSE); curl_setopt(\$ch, CURLOPT_SSL_VERIFYPEER, FALSE); curl_setopt(\$ch, CURLOPT_HTTPHEADER, &amp;\$headers); curl_setopt(\$ch, CURLOPT_URL, \$url); curl_setopt(\$ch, CURLOPT_WRITEFUNCTION, write_callback); curl_setopt(\$ch, CURLOPT_POSTFIELDS, \$data); curl_setopt(\$ch, CURLOPT_VERBOSE, 1);  \$result = curl_exec(\$ch);  \$data = &lt;&lt;&lt;EOF &lt;...&gt; EOF;  sendCommand(\$operator, \$data, \$login, \$passwd, \$host, \$port);</pre>	<p>Send response packet to stdout.</p>
	Send request packet to Expand server
	Specify Plesk Expand API endpoint
	Create HTTP message-header
	Initialize CURL engine
	Send data to Expand server and request output.
	Specify request packet content
	Call previously described function.

## Perl Client

The following code sample presents ready-to-use client application written in Perl. This application interacts with the Expand server via the external endpoint. The client requests information on Plesk servers managed by a certain Expand server, and processes the server response. To perform custom operations, substitute the packet contents.

**Note:** If you want to use this sample, make sure you have Perl 5. For more information, visit [www.perl.org](http://www.perl.org).

```
#!/usr/bin/perl
use strict;
use Data::Dumper;
use Net::SSLeay qw(get_https make_form make_headers post_https);
$Net::SSLeay::ssl_version = 3;
my $host = "expand-example.com";
my $port = 8442;
my $login = 'root';
my $script = "/webgate.php";
my $passwd = 'setup';
my $operator_name = 'exp_plesk_domain';
my $request = <<EOL;

<?xml version="1.0" encoding="UTF-8"?>
<packet version="2.0.1.2">
  <get>
    <filter></filter><!-- select all Plesk servers -->
    <dataset>
      <gen_info></gen_info>
    </dataset>
  </get>
</packet>
EOL

my $headers = make_headers (
  'HTTP_AUTH_LOGIN'=> $login,
  'HTTP_AUTH_PASSWD' => $passwd,
  'HTTP_AUTH_OP'=> $operator_name,
  'Expect:',
);
print "----- REQUEST -----<n";
print "REQUEST_HEADERS=".Dumper($headers)."<n";
print "REQUEST_DATA<n".<request.<n";
print "----- RESULT -----<n";
my ($reply_data, $reply_type, %reply_headers) =
  post_https($host, $port, $script, $headers, $request, 'text/xml');
print "REPLY_HEADERS=".Dumper(\%reply_headers)."<n";
print "REPLY_TYPE=".<reply_type."<n";
print "REPLY_DATA:<n".<reply_data."<n";
```

To tailor this application to your needs, replace the following variable values with the appropriate content:

Variable	Description
my \$host	Expand host name
my \$port	Expand port number (Default: 8442)
my \$login	Expand Administrator login
my \$passwd	Expand Administrator password
my \$operator_name	Operator name
my \$request	Request packet contents

This table explains the client application code.

Operation / group of operations	Action performed
<pre>use Data::Dumper; use Net::SSLeay qw(get_https make_form make_headers post_https); \$Net::SSLeay::ssl_version = 3;  my \$request = &lt;&lt;EOL; &lt;...&gt; EOL  my \$script = "/webgate.php";  my \$headers = make_headers ( 'HTTP_AUTH_LOGIN'=&gt; \$login, 'HTTP_AUTH_PASSWD' =&gt; \$passwd, 'HTTP_AUTH_OP'=&gt; \$operator, 'Expect:', );  print "----- REQUEST -----\n"; print "REQUEST_HEADERS=".Dumper(\$headers).""; print "REQUEST_DATA\n".\$request.\n"; print "----- RESULT -----\n";  my (\$reply_data, \$reply_type, %reply_headers) = post_https(\$host, \$port, \$script, \$headers, \$request, 'text/xml');  print "REPLY_HEADERS=".Dumper(\%reply_headers).\n"; print "REPLY_TYPE=".\$reply_type.\n"; print "REPLY_DATA:\n".\$reply_data."";</pre>	<p>Call up necessary libraries</p> <p>Specify request packet content</p> <p>Specify Expand endpoint URL</p> <p>Create HTTP message-header</p> <p>Send request packet to stdout (optional)</p> <p>Send request packet to Expand server</p> <p>Send response packet to stdout</p>