
WebFaction API Documentation

Release

December 22, 2011

CONTENTS

1	Introduction	3
2	Tutorial	5
2.1	Getting Started	5
2.2	Creating an Email Address	5
2.3	Installing an Application	6
2.4	Packaging the Install Script for the Control Panel	7
2.5	Additional Resources	8
3	API Reference	9
3.1	General	9
3.2	Email	9
3.3	Websites and Domains	13
3.4	Applications	15
3.5	Cron	16
3.6	DNS	16
3.7	Databases	17
3.8	Files	18
3.9	Shell Users	19
3.10	Servers	20
3.11	Miscellaneous	20
	Index	23

Contents:

INTRODUCTION

The WebFaction API (Application Programming Interface) is a powerful [XML-RPC](#) interface for managing many control panel and account tasks. With the WebFaction API, you can automate application installation, email address configuration, and more.

Like other XML-RPC APIs, the WebFaction API works by sending a short piece of XML over HTTP. Luckily, many languages have XML-RPC libraries to make requests quick and painless.

For example, you can send an XML-RPC request using Python's `xmlrpclib` module:

```
>>> import xmlrpclib
>>> server = xmlrpclib.ServerProxy('https://api.webfaction.com/')
>>> session_id, account = server.login('widgetsco', 'widgetsrock')
```

Or with Ruby's `xmlrpc` package:

```
>> require 'xmlrpc/client'
=> true
>> require 'pp'
=> true
>> server = XMLRPC::Client.new2("https://api.webfaction.com/")
#<XMLRPC::Client:0x5b1698 @cookie=nil, @create=nil, @port=443>
>> pp server.call("login", "widgetsco", "widgetsrock")
["ca4c008c24c0de9c9c8",
 {"mail_server"=>"Mail5",
  "web_server"=>"Web55",
  "username"=>"widgetsco",
  "id"=>687,
  "home"=>"/home"}]
=> nil
```

To learn more about XML-RPC and find an implementation in your favorite language, please visit [XMLRPC.com](#). ..
include:: common.subs

TUTORIAL

The WebFaction API allows you to write scripts to automate certain tasks that you would normally accomplish with the control panel or an SSH session.

For instance, you could use the API to write a script to configure lots of email addresses, instead of creating them one by one in the control panel.

You can also use the API to automate the installation of any application that you like, and you can share the resulting install script to allow other users to use it in one click.

2.1 Getting Started

The API is a set of methods available via XML-RPC calls at the URL `https://api.webfaction.com/`. In this documentation we will use the Python programming language to talk to the API, but you can use any language that you want, provided that it has an XML-RPC library.

First, connect to the server and login:

```
>>> import xmlrpclib
>>> server = xmlrpclib.ServerProxy('https://api.webfaction.com/')
>>> session_id, account = server.login('test5', 'password')
>>> account
{'username': 'test5', 'home': '/home2', 'id': 237}
```

The username and password passed to the login method are those used to login to the control panel.

As you can see, the `login()` method returns a tuple. The first element is a string containing a session ID that you will need to pass to all subsequent methods. The second element is a dictionary containing various data about your account, including the base home directory (usually `/home` or `/home2`).

2.2 Creating an Email Address

Let's create a new email address for our account:

```
>>> server.create_email(session_id, 'user@mytestdomain.com', 'test5')
{'email_address': 'user@mytestdomain.com', 'id': 2037, 'targets': 'test5'}
```

The `create_email()` method takes the session ID, the email address and a string of comma separated target mailboxes. It returns a dictionary containing various data about the newly created email address.

This call is equivalent to creating the email address from the control panel, but the advantage is that you can script it.

2.3 Installing an Application

Now let's see which methods we can use to install an application. For this tutorial, let's install a Joomla application.

First, create a `Static/CGI/PHP` application:

```
>>> server.create_app(session_id, 'my_joomla_app', 'static', False, '')
{'app_type': 'static',
 'name': 'my_joomla_app',
 'id': 3901,
 'autostart': False,
 'port': 0,
 'extra_info': ''}
```

`create_app()` uses the following parameters:

- *session_id* – session ID returned by `login()`
- *name* – name of the application
- *type* – type of the application. This can be any of the types supported by the control panel.
- *autostart* (boolean) – whether the app should get restarted with an `autostart.cgi` script
- *extra_info* – additional information required by the application (for example, a file path). If `extra_info` is not required or used by the application, it is ignored.

Calling `create_app()` is equivalent to creating the application through the control panel: it creates the directory and configures everything that's needed for this application.

Next, you need to download the Joomla tarball and untar it in the app directory. To do so we'll use the `system()` method of the API, which allows us to execute some command on our server as if we were logged in with SSH. Since the default Static/CGI/PHP app comes with an `index.html` file which would shadow the `index.php` file provided by Joomla, we will also delete that file:

```
>>> cmd = "rm -f index.html;"
>>> cmd += "wget https://wiki.webfaction.com/attachment/wiki/JoomlaFiles/Joomla_1.5.0-Beta-Full_Packa
>>> cmd += "tar xzvf Joomla_1.5.0-Beta-Full_Package.tar.gz?format=raw"
>>> server.system(session_id, cmd)
''
```

Because we previously installed an application, the `system()` method automatically runs in the new application's directory. The `system()` method returns whatever the command printed to standard output. If the command prints something to standard error, `system()` raises an error with that text.

Next, create a MySQL database, since Joomla requires one:

```
>>> server.create_db(session_id, 'test5_joomla_db', 'mysql', 'db_password')
{'type': 'mysql',
 'id': '1161011151160530951061111111091080970951000980451',
 'name': 'test5_joomla_db'}
```

This creates a MySQL database called `test5_joomla_db` and a user of the same name, with the password `db_password`.

Next, copy the Joomla config file from `configuration.php-dist` to `configuration.php` and then edit it to specify the database connection settings:

```
>>> server.system(session_id, "cp configuration.php-dist configuration.php;")
''
```

```
>>> server.replace_in_file(session_id, 'configuration.php',
    ("var $user = '';", "var $user = 'test5_joomla_db';"),
    ("var $password = '';", "var $password = 'db_password';"),
    ("var $db = '';", "var $db = 'test5_joomla_db';"))
''
```

The handy `replace_in_file()` method lets us find and replace in a file. It takes a session ID, the name of the file, and any number of tuples containing a string to replace and the replacement string.

There are a few other steps needed to install a Joomla application: we need to edit the Joomla SQL file and run it. We won't detail them here—you can look at the [actual script](#) for details.

2.4 Packaging the Install Script for the Control Panel

Previously, we installed a Joomla application by manually running a bunch of commands. We can package these commands in an install script for the control panel to run for us. The advantage is that we can then run this install script over and over again directly from the control panel or share it with others.

2.4.1 How the Install Script is Run

To use an install script in the control panel:

1. Log in to the control panel.
2. Under *Domains / websites*, click *Applications*.
3. Click the *Add new* button.
4. In the *Name* field, enter an application name.
5. In the *App category* menu, click to select *Custom*. The *Install script url* field will appear.
6. Load the install script.

Load the script from a URL:

- (a) In the *Install script url* field, enter the URL of the install script.
- (b) Click *Fetch URL*.
- (c) Optionally, click *View/hide script* to see the loaded script.

Copy and paste the script into the form:

- (a) Next to the *Install script url* field, click *View/hide script*. The *Install script* field will appear.
 - (b) Copy and paste your script code into the *Install script* field.
7. If needed, select the *Autostart* checkbox.
 8. If needed, enter text into the *Extra info* field.
 9. Click the *Create* button.

When you click the *Create* button, the script will call the script with the following parameters:

```
install_script create|delete username password app_name autostart extra_info
```

- *username* – control panel username
- *password* – user's hashed password

- *app_name* – application name from *Name* field.
- *autostart* (boolean) – whether the user selected the *Autostart* checkbox
- *extra_info* – contents of the *Extra info* field.

When the user creates the app, the control panel will call the script with `create` as the first parameter. If the user deletes the app later on, the control panel will call the script with `delete` as the first parameter.

When it calls the script with `create`, the control panel expects the script to print the application ID to standard output and nothing else. If the script prints anything else to standard output, or prints anything to standard error, the control panel will display it as an error message on the add application page.

When it calls the script with `delete`, the control panel expects your script to not print anything to standard output or standard error. If anything gets printed the control panel will display it as an error message.

Additionally, if your script is written in Python and you include a docstring ([PEP 257](#)) at the beginning of the script, the control panel will use it as the application documentation.

To see two examples of install scripts, take a look at the [Joomla install script](#) or the [MoinMoin install script](#).

2.4.2 Making a Script Available in One Click

Now that you have your install script, one way to share it would be to ask others copy and paste it in the control panel. That's not convenient but fortunately, there is a better way.

On the add application page in the control panel, there is a field *Install script url* that appears when you select *Custom install script*. If you enter a URL and click *Fetch URL*, the control panel will look for an install script at that URL. For the control panel to find the script on the page, your script must be enclosed between these two magic tags:

```
-----BEGIN WEBFACTION INSTALL SCRIPT-----  
-----END WEBFACTION INSTALL SCRIPT-----
```

If you make your script available on the web, all you have to do to let anyone use it is give them a special URL in the form `'https://my.webfaction.com/app/create?script_url=location'` where *location* is the escaped URL to your script.

You can see examples at [InstallScripts](#), where Joomla, MoinMoin, and other applications can be installed using that approach.

2.5 Additional Resources

For more information about the WebFaction API, please consult the [API Reference](#).

API REFERENCE

The WebFaction XML-RPC API provides methods to make many control panel tasks (and others) scriptable.

This documentation is a complete reference to all of the possible API calls. Please note that all arguments are positional and many are required. Arguments may be omitted if they have a default value (for example, `arg_name=some_value`) and follow any other arguments with which you have provided a value. The XML-RPC standard does not support keyword arguments.

3.1 General

login (*username, password, machine=None*)

Log in a user and return credentials.

Parameters

- **username** – a valid WebFaction control panel username
- **password** – a valid WebFaction control panel user’s password
- **machine** – the case-sensitive name of the machine to access (for example, `Web55`). This parameter applies to accounts with plans on multiple machines. The parameter may be omitted for accounts on one machine only.

The values returned are a session ID and a dictionary. The dictionary contains the following key-value pairs:

id account ID

username username

home home directory path

web_server Web server associated with the logged in account (for example, `Web55`)

mail_server mail server associated with the logged in account (for example, `Mailbox2`)

Note: The session ID is required for all subsequent API calls.

3.2 Email

change_mailbox_password (*session_id, mailbox, password*)

Change a mailbox password.

Parameters

- **session_id** – session ID returned by `login()`
- **mailbox** – a valid mailbox name
- **password** – the new mailbox password

See Also:

See [Strengthening Passwords](#) for important information about choosing passwords.

```
create_email ( session_id, email_address, targets, autoresponder_on=False, autoresponder_subject='',
               autoresponder_message='',      autoresponder_from='',      script_machine='',
               script_path='')
```

Create an email address which delivers to the specified mailboxes.

If `autoresponder_on` is `True`, then an autoresponder subject, message, and from address may be specified.

Parameters

- **session_id** – session ID returned by `login()`
- **email_address** – an email address (for example, `name@mydomain.tld`)
- **targets** (*string*) – names of destination mailboxes or addresses, separated by commas
- **autoresponder_on** (*boolean*) – whether an autoresponder is enabled for the address
- **autoresponder_subject** – subject line of the autoresponder message
- **autoresponder_message** – body of the autoresponder message
- **autoresponder_from** – originating address of the autoresponder message
- **script_machine** – a machine name for specifying a path to a script
- **script_path** – a path to a script

See Also:

```
update_email()
```

```
create_mailbox ( session_id, mailbox, enable_spam_protection=True, discard_spam=False,
                 spam_redirect_folder='', use_manual_procmairc=False, manual_procmairc='')
```

Create a mailbox and return a dictionary containing the following key-value pairs:

id mailbox ID

name name of mailbox created

enable_spam_protection a boolean indicating whether spam protection is enabled for the mailbox

password a generated password

discard_spam a boolean indicating whether spam emails should be discarded

spam_redirect_folder a string containing the path to an IMAP folder where the server stores spam messages

use_manual_procmairc a boolean indicating whether manual procmairc rules are enabled

manual_procmairc a string containing the manual procmairc rules

Parameters

- **session_id** – session ID returned by `login()`
- **mailbox** – name of mailbox

- **enable_spam_protection** (*boolean*) – whether spam protection is enabled for the new mailbox
- **discard_spam** (*boolean*) – whether spam received by the new mailbox will be discarded. Warning: setting this to True may cause you to lose emails forever.
- **spam_redirect_folder** – name of the folder where email identified as spam is be stored
- **use_manual_procmailrc** (*boolean*) – whether a manual procmailrc file is to be used for the new mailbox
- **manual_procmailrc** – the procmailrc rules for the new mailbox

See Also:

`update_mailbox()`

delete_email (*session_id*, *email_address*)

Delete an email address.

Parameters

- **session_id** – session ID returned by `login()`
- **email_address** – an email address (for example, `name@mydomain.tld`)

delete_mailbox (*session_id*, *mailbox*)

Delete a mailbox.

Parameters

- **session_id** – session ID returned by `login()`
- **mailbox** – name of mailbox

list_emails (*session_id*)

Return a list of dictionaries with the following key-value pairs:

id an email ID

email_address an email address (for example, `name@mydomain.tld`)

targets mailboxes or email addresses to which the address is set to deliver

autoresponder_on a boolean indicating whether an autoresponder is enabled for the address

autoresponder_subject the autoresponder subject line (if applicable)

autoresponder_message the autoresponder message body (if applicable)

autoresponder_from the autoresponder from address (if applicable)

Parameters **session_id** – session ID returned by `login()`

list_mailboxes (*session_id*)

Return a list of dictionaries with the following key-value pairs:

id a mailbox ID

mailbox the name of mailbox

enable_spam_protection a boolean indicating whether spam protection is enabled for the mailbox

discard_spam a boolean indicating whether spam emails are discarded

spam_redirect_folder a string containing the path to an IMAP folder where the server stores spam messages

use_manual_procmailrc a boolean indicating whether you want to use your own .procmailrc file

manual_procmailrc a string containing the text that should go in the .procmailrc file

Parameters *session_id* – session ID returned by `login()`

```
update_email ( session_id, email_address, targets, autoresponder_on=False, autoresponder_subject='',
               autoresponder_message='', autoresponder_from='', script_machine='',
               script_path='' )
```

Change an existing email address. The email address must exist. A dictionary containing the following key-value pairs will be returned:

id email ID

email_address email address (for example, `name@mydomain.tld`)

targets mailboxes or email addresses to which the address is set to deliver

Parameters

- **session_id** – session ID returned by `login()`
- **email_address** – an email address (for example, `name@mydomain.tld`)
- **targets** (*list, tuple*) – names of destination mailboxes
- **autoresponder_on** (*boolean*) – whether an autoresponder is enabled for the address
- **autoresponder_subject** – subject line of the autoresponder message
- **autoresponder_message** – body of the autoresponder message
- **autoresponder_from** – originating address of the autoresponder message
- **script_machine** – a machine name for specifying a path to a script
- **script_path** – a path to a script

See Also:

`create_email()`

```
update_mailbox ( session_id, mailbox, enable_spam_protection=True, discard_spam=False,
                 spam_redirect_folder='', use_manual_procmailrc=False, manual_procmailrc='' )
```

Change an existing mailbox. The mailbox must exist.

Parameters

- **session_id** – session ID returned by `login()`
- **mailbox** – name for new mailbox
- **enable_spam_protection** (*boolean*) – whether spam protection is enabled for this mailbox
- **discard_spam** (*boolean*) – whether spam received by this mailbox will be discarded
- **spam_redirect_folder** – name of the folder where email identified as spam is be stored
- **use_manual_procmailrc** (*boolean*) – whether a manual procmailrc setting is to be used for this mailbox
- **manual_procmailrc** – the procmailrc rules for this mailbox

See Also:

`create_mailbox()`

3.3 Websites and Domains

create_domain (*session_id*, *domain*, **subdomains*)

Create a domain entry. If *domain* is an existing domain, subdomains can be created with subsequent parameters.

Parameters

- **session_id** – session ID returned by `login()`
- **domain** – a domain name in the form `mydomain.tld`

Example:

Create a domain entry for `widgetcompany.biz`:

```
>>> import xmlrpclib
>>> server = xmlrpclib.ServerProxy('https://api.webfaction.com/')
>>> session_id, account = server.login('widgetsco', 'widgetsrock')
>>> server.create_domain(session_id, 'widgetcompany.biz', 'www', 'design')
{'domain': 'widgetcompany.biz',
 'id': 47255,
 'subdomains': ['www', 'design']}
```

create_website (*session_id*, *website_name*, *ip*, *https*, *subdomains*, **site_apps*)

Create a new website entry.

Parameters

- **session_id** – session ID returned by `login()`
- **website_name** – the name of the new website entry
- **ip** – IP address of the server where the entry resides
- **https** (*boolean*) – whether the website entry should use a secure connection
- **subdomains** – a list of the (sub)domains to be associated with the website entry.
- **site_apps** – parameters after *subdomains* are tuples or lists of a valid application name and a path (for example, `('htdocs', '/')` or `['mydjangoapp', '/myapp/']`).

Example:

Create a website entry for `widgetcompany.biz`'s new Django project over HTTPS:

```
>>> import xmlrpclib
>>> server = xmlrpclib.ServerProxy('https://api.webfaction.com/')
>>> session_id, account = server.login('widgetsco', 'widgetsrock')
>>> server.create_website(session_id,
... 'widgets_on_the_web',
... '174.133.82.194',
... True,
... ['widgetcompany.biz', 'www.widgetcompany.biz'],
... ['django', '/'])
{'https': True,
 'id': 67074,
 'ip': '174.133.82.194',
 'name': 'widgets_on_the_web',
 'site_apps': [['django', '/']],
 'subdomains': ['widgetcompany.biz', 'www.widgetcompany.biz']}
```

update_website (*session_id*, *website_name*, *ip*, *https*, *subdomains*, **site_apps*)

Update a new website entry.

Parameters

- **session_id** – session ID returned by `login()`
- **website_name** – the name of the new website entry you want to update; this parameter can't be updated
- **ip** – IP address of the server where the entry resides
- **https** (*boolean*) – whether the website entry you want to update uses a secure connection; this parameter can't be updated
- **subdomains** – a list of the (sub)domains to be associated with the website entry.
- **site_apps** – parameters after *subdomains* are tuples or lists of a valid application name and a path (for example, `('htdocs', '/')` or `['mydjangoapp', '/myapp/']`).

Example:

Update a website entry for widgetcompany.biz's new Django project over HTTPS:

```
>>> import xmlrpclib
>>> server = xmlrpclib.ServerProxy('https://api.webfaction.com/')
>>> session_id, account = server.login('widgetsco', 'widgetsrock')
>>> server.update_website(session_id,
... 'widgets_on_the_web',
... '174.133.82.195',
... True,
... ['widgetcompany.biz', 'dev.widgetcompany.biz'],
... ('django', '/'), ('wordpress', '/blog'))
{'https': True,
 'id': 67074,
 'ip': '174.133.82.195',
 'name': 'widgets_on_the_web',
 'site_apps': [['django', '/'], ['wordpress', '/blog']],
 'subdomains': ['widgetcompany.biz', 'dev.widgetcompany.biz']}
```

delete_domain (*session_id, domain, *subdomains*)

Delete a domain record or subdomain records.

If subdomains are specified with parameters after *domain*, delete only those subdomains and not the *domain* entry.

Parameters

- **session_id** – session ID returned by `login()`
- **domain** – name of the domain to be deleted or the parent domain of the subdomains to be deleted

delete_website (*session_id, website_name, ip=None, https=None*)

Delete a website entry.

Parameters

- **session_id** – session ID returned by `login()`
- **website_name** – name of website entry to be deleted
- **ip** – IP address of the server where the entry resides
- **https** (*boolean*) – whether the entry uses a secure connection

list_domains (*session_id*)

Return a list of dictionaries with the following key-value pairs:

id domain ID

domain domain (for example, `mydomain.tld`)

subdomains list of subdomains for that domain (for example, `["www", "www2"]`)

Parameters `session_id` – session ID returned by `login()`

`list_websites (session_id)`

Return a list of dictionaries with the following key-value pairs:

id website ID

name website name

ip website IP address

https whether the website is available over https

subdomains list of subdomains for that website

website_apps list of apps and their paths served by that website; each item in the list is a 2-item list [`<app name>`, `path`]

Parameters `session_id` – session ID returned by `login()`

3.4 Applications

`create_app (session_id, name, type, autostart, extra_info, script_code='')`

Create a new application.

Parameters

- **session_id** – session ID returned by `login()`
- **name** – name of the application
- **type** – type of the application (for example, `django102_mp331_25` or `static`)
- **autostart** (*boolean*) – whether the app should restart with an `autostart.cgi` script
- **extra_info** – additional information required by the application (for example, a file path). If `extra_info` is not required or used by the application, it is ignored.

Note: For a complete list of possible applications and their associated `extra_info` parameters, please see the [Home > App types](#) (login required) page on the WebFaction control panel.

`delete_app (session_id, name)`

Delete an application.

Parameters

- **session_id** – session ID returned by `login()`
- **name** – name of the application

`list_apps (session_id)`

Return a list of dictionaries with the following key-value pairs:

id app ID

name app name

type app type

autostart whether the app uses autostart or not

port contains the port number if the app listens on a port; contains 0 otherwise

extra_info contains the extra info for the app if any

machine name of the machine where the app resides

Parameters **session_id** – session ID returned by `login()`

3.5 Cron

`create_cronjob (session_id, line)`

Create a new cron job.

Parameters

- **session_id** – session ID returned by `login()`
- **line** – crontab line to be added

See Also:

For more information about the cron syntax, please see the Wikipedia entry on [cron](#).

`delete_cronjob (session_id, line)`

Remove an existing cron job.

Parameters

- **session_id** – session ID returned by `login()`
- **line** – crontab line to be removed

3.6 DNS

`create_dns_override (session_id, domain, a_ip='', cname='', mx_name='', mx_priority='',
spf_record='')`

Create a DNS override entry.

Parameters

- **session_id** – session ID returned by `login()`
- **domain** – domain name to be overridden (for example, `sub.mydomain.net`)
- **a_ip** – A IP address
- **cname** – CNAME record
- **mx_name** – Mail exchanger record host name
- **mx_priority** – Mail exchanger record priority
- **spf_record** – TXT record

delete_dns_override (*session_id*, *domain*, *a_ip*='', *cname*='', *mx_name*='', *mx_priority*='', *spf_record*='')

Delete a DNS override entry and return a list of deleted entries.

Parameters

- **session_id** – session ID returned by `login()`
- **domain** – domain name override to be deleted (for example, `sub.mydomain.net`)
- **a_ip** – A IP address
- **cname** – CNAME record
- **mx_name** – Mail exchanger record host name
- **mx_priority** – Mail exchanger record priority
- **spf_record** – TXT record

list_dns_overrides (*session_id*)

Return a list of dictionaries with the following key-value pairs:

id domain ID

domain domain name to be overridden (for example, `sub.mydomain.net`)

a_ip A IP address

cname CNAME record

mx_name Mail exchanger record host name

mx_priority Mail exchanger record priority

spf_record TXT record

srv_record Service record

Parameters **session_id** – session ID returned by `login()`

3.7 Databases

create_db (*session_id*, *name*, *db_type*, *password*)

Create a database.

Parameters

- **session_id** – session ID returned by `login()`
- **name** – database name
- **db_type** – database type (`mysql` or `postgresql`)
- **password** – password for the default database user

Note: *name* must be the username used to log in or begin with the username and an underscore. MySQL database names may not exceed 16 characters, including the username prefix.

delete_db (*session_id*, *name*, *db_type*)

Delete a database.

Parameters

- **session_id** – session ID returned by `login()`
- **name** – database name
- **db_type** – database type (`mysql` or `postgresql`)

list_dbs (*session_id*)

Return a list of dictionaries with the following key-value pairs:

id database ID

name database name

db_type database type (`mysql` or `postgresql`)

machine name of the machine where the app resides

Parameters **session_id** – session ID returned by `login()`

3.8 Files

replace_in_file (*session_id, filename, *changes*)

Find and replace a string in a file in the users’s home directory tree.

Parameters

- **session_id** – session ID returned by `login()`
- **filename** – path to file from the user’s home directory

Any parameters after *filename* are tuples or lists containing a pair of strings where the first is the string to be replaced and the second is the replacement text.

Example:

Find all appearances of the word “eggs” in a file in the user’s home directory and replace them with the word “spam.”

```
$ cat myfile.txt
eggs, spam, spam, and spam.
spam, spam, spam, and eggs.
```

```
>>> import xmlrpclib
>>> server = xmlrpclib.ServerProxy('https://api.webfaction.com/')
>>> session_id, account = server.login('widgetesco', 'widgetrock')
>>> server.replace_in_file(sid, 'myfile.txt', ('eggs', 'spam'))
''
>>> exit()
```

```
$ cat myfile.txt
spam, spam, spam, and spam.
spam, spam, spam, and spam.
```

write_file (*session_id, filename, str, mode = 'wb'*)

Write a string to a file in the user’s home directory tree.

Parameters

- **session_id** – session ID returned by `login()`
- **filename** – path to file to be written from the user’s home directory.

- **str** – string to be written
- **mode** – write mode

Note: Commonly, the write mode is `w` for plain text and `wb` for binaries. `a` and `ab` can be used to append to files.

See Also:

For more information about write modes, please see `open()`.

3.9 Shell Users

list_users (*session_id*)

Return a list of shell users represented as dictionaries with the following key-value pairs:

username username

machine the user's server (for example, `Web100`)

shell the user's configured command line interpreter (for example, `bash` or `tcsh`)

groups a list of extra permissions groups of which the user is a member

Parameters **session_id** – session ID returned by `login()`

change_user_password (*session_id, username, password*)

Change a shell user's password and return a dictionary of the new user in the manner of `list_users()` .

Parameters

- **session_id** – session ID returned by `login()`
- **username** – username
- **password** – a new password

create_user (*session_id, username, shell, groups*)

Create a new shell user and return a dictionary of the new user in the manner of `list_users()` .

Parameters

- **session_id** – session ID returned by `login()`
- **username** – username
- **shell** – the user's command line interpreter. Acceptable values are `none`, `bash`, `sh`, `ksh`, `csh`, or `tcsh` . A user with `none` as the configured shell has FTP access, but does not have SSH access.
- **groups** – a list of extra permissions groups of which the new user is to be a member. All users are automatically a member of their own group; do not include the user's own group in the list. Use an empty list to specify no extra groups.

delete_user (*session_id, username*)

Delete user `username` and return a dictionary of the deleted user in the manner of `list_users()` .

Parameters

- **session_id** – session ID returned by `login()`

- **username** – username

3.10 Servers

list_machines (*session_id*)

List all of the machines associated with the account. The method returns a list of dictionaries with the following key-value pairs:

machine A system's name (for example, `Web100`).

id The system's identifying number.

operating_system The machine's operating system (for example, `Centos6-64bit`).

location The machine's location (for example, `USA`).

list_ips (*session_id*)

List all of the machines associated with the account and their IP addresses. This method returns a list of dictionaries with the following key-value pairs:

name A system's name (for example, `Web100`).

ip The system's IP address.

is_main A `True` or `False` value indicating whether the IP address is the primary IP address for the server (`True`) or an extra IP address provisioned to the account (`False`).

id The IP's identifying number.

3.11 Miscellaneous

run_php_script (*session_id*, *script*, *code_before*='')

Run PHP code followed by a PHP script. The PHP code and script is run as the user.

Parameters

- **session_id** – session ID returned by `login()`
- **script** – an absolute path to script to be run or the path to the script starting from the user's home directory
- **code_before** – PHP code to be executed before *script*

set_apache_acl (*session_id*, *paths*, *permission*='rwx', *recursive*=False)

Grant the machine-wide Apache instance access to files or directories.

Parameters

- **session_id** – session ID returned by `login()`
- **paths** (*string*, *list*) – path from home directory
- **permission** – `r`, `w`, or `x` or a combination thereof
- **recursive** (*boolean*) – whether Apache's access is granted recursively

system (*session_id*, *cmd*)

Execute a command as the user, as if through SSH. If an application was installed previously in the session, the command will be run from the directory where that application was installed.

Note: If *cmd* prints anything to standard error, the API method will return an XML-RPC fault. Many XML-RPC libraries will express the fault as an exception.

Parameters

- **session_id** – session ID returned by `login()`
- **cmd** – a shell command to be executed
- *genindex*

INDEX

C

change_mailbox_password(), 9
change_user_password(), 19
create_app(), 15
create_cronjob(), 16
create_db(), 17
create_dns_override(), 16
create_domain(), 13
create_email(), 10
create_mailbox(), 10
create_user(), 19
create_website(), 13

D

delete_app(), 15
delete_cronjob(), 16
delete_db(), 17
delete_dns_override(), 16
delete_domain(), 14
delete_email(), 11
delete_mailbox(), 11
delete_user(), 19
delete_website(), 14

L

list_apps(), 15
list_dbs(), 18
list_dns_overrides(), 17
list_domains(), 14
list_emails(), 11
list_ips(), 20
list_machines(), 20
list_mailboxes(), 11
list_users(), 19
list_websites(), 15
login(), 9

P

Packaging, 7
Python Enhancement Proposals
 PEP 257, 8

R

replace_in_file(), 18
run_php_script(), 20

S

set_apache_acl(), 20
system(), 20

U

update_email() (built-in function), 12
update_mailbox(), 12
update_website(), 13

W

write_file(), 18