**JUNIPER NETWORKS,INC.**

**NETCONF PHP TOOLKIT**

**sgambhir**

**NETCONF PHP TOOLKIT**

A step-by-step guide on using the PHP API’s

TABLE OF CONTENTS

1. Abstract
2. How Netconf PHP Toolkit works
3. Installation Instructions
4. What is?
   1. A Device
   2. A XML
   3. A XMLBuilder
5. How to?
   1. Create and use a Device
   2. Create and build upon an XML
   3. Parse the RPC Reply
6. Lets learn with examples
   1. Web-based CLI(works for operational mode commands only)
7. Feedback
8. Abstract

Each Juniper Networks routing platform running JUNOS Internet software release 7.5B1 or later supports the NETCONF API. The NETCONF API provides mechanisms to install, manipulate and delete the configuration of network devices. The NETCONF API uses an Extensible markup Language (XML) based data encoding for the configuration data as well as operations and messages defined in the API.

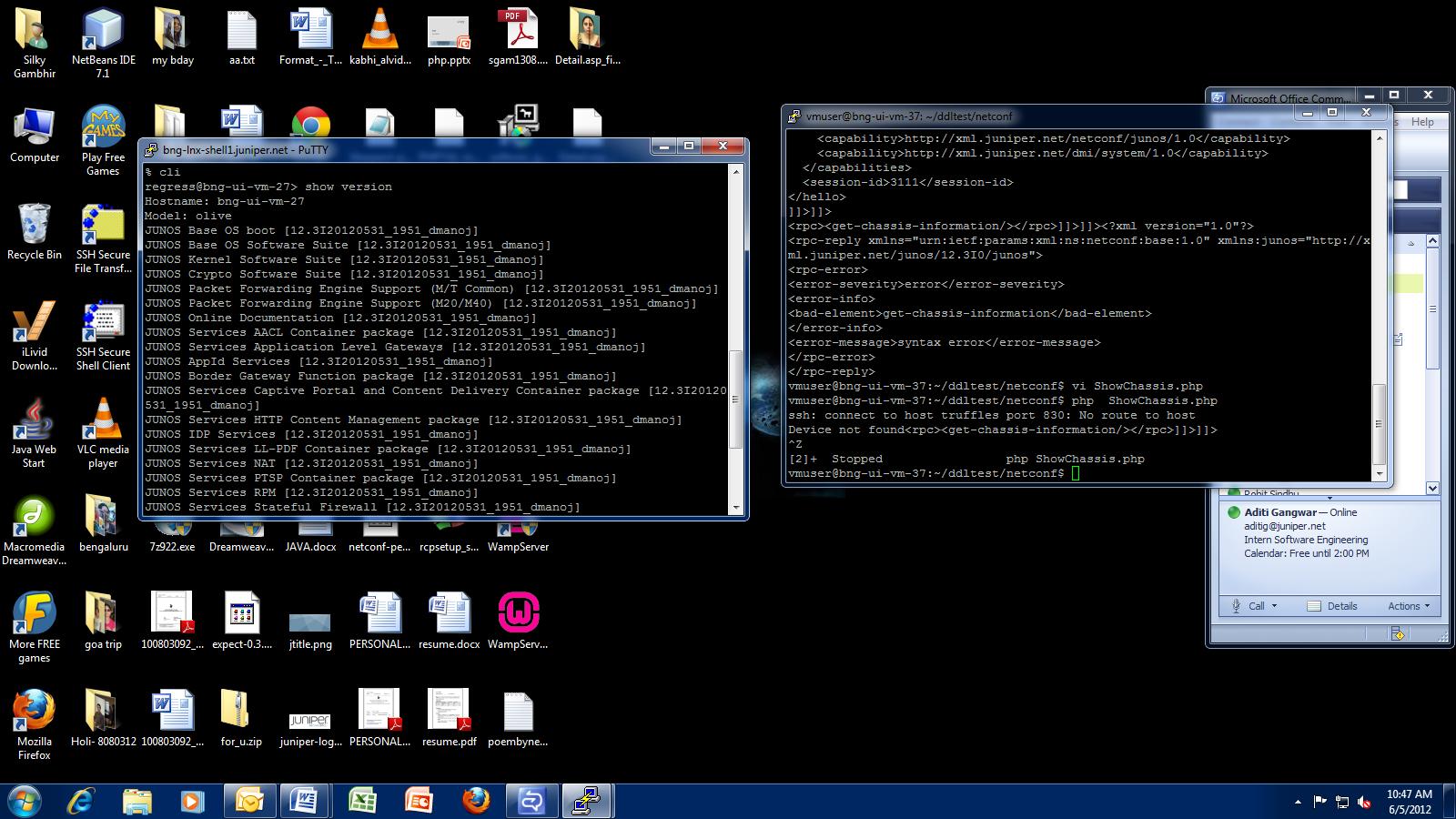
To know more about the NETCONF Configuration Protocol, please visit http://tools.ietf.org/html/rfc4741 The NETCONF Java toolkit provides an object-oriented interface for communicating with the NETCONF server so you can start using the NETCONF API quickly and easily.

1. How NETCONF PHP Toolkit Works

The communication between the Management server(For example, your Desktop) and the NETCONF Server(For example, a JUNOS device), via the Java NETCONF toolkit can be split into three components:

1. **Setup a NETCONF Session over SSH-2, between the Management server and NETCONF server.**
2. **Create an ‘RPC’ corresponding to your request and send this over to the NETCONF server.**
3. **Take action on the RPC Reply received from the NETCONF server.**

Lets understand this with an example. Alex wants to find out software information of a JUNOS box (Equivalent to running “show version” on JUNOS CLI”).



*Figure 2.1 Running “show version on JUNOS CLI*

He wants to use the NETCONF PHP toolkit to achieve the same result. Following are the steps to do this:

1. **Install the PHP Netconf Toolkit** (see Installation Instructions).
2. **Create a .php file (lets name it ShowVersion.php)** and write the code inside the file, following the below instructions.
3. **Create a Device object to create a ‘default’ NETCONF Session over SSH-2 with the NETCONF server.**

*//Code*

*$my\_device = new Device(“hostname”,”user”,”password”);*

*try {*

*$my\_device->connect();*

*}*

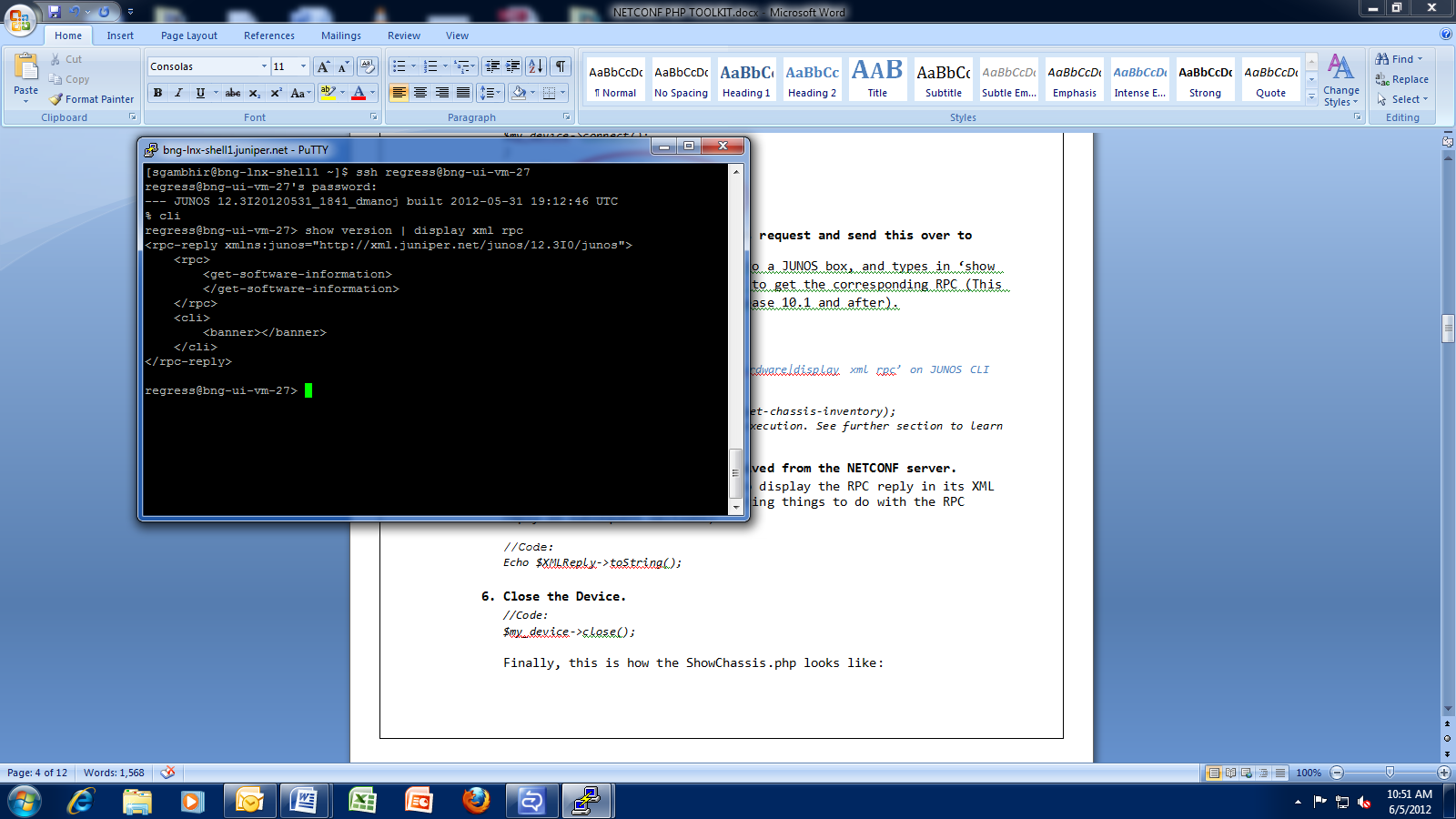
*catch(Exception $e) {*

*echo $e->getMessage();*

*}*

1. **Create an RPC corresponding to his request and send this over to NETCONF server.**

To figure out the RPC, he logins to a JUNOS box, and types in ‘show chassis hardware|display xml rpc’ to get the corresponding RPC (This feature is available in JUNOS release 10.1 and after).



*Figure 2.2: Running ‘show version | display xml rpc’ on JUNOS CLI*

*//Code*

*$XMLReply = $my\_device->executeRPC(“get-software-information”);*

*//This is the simplest possible RPC execution. See further section to learn how to execute complex RPC’s.*

1. **Take action on the RPC Reply received from the NETCONF server.**

In this case, Alex simply wants to display the RPC reply in its XML form. (We’ll look at more interesting things to do with the RPC reply in subsequent sections)

*//Code:*

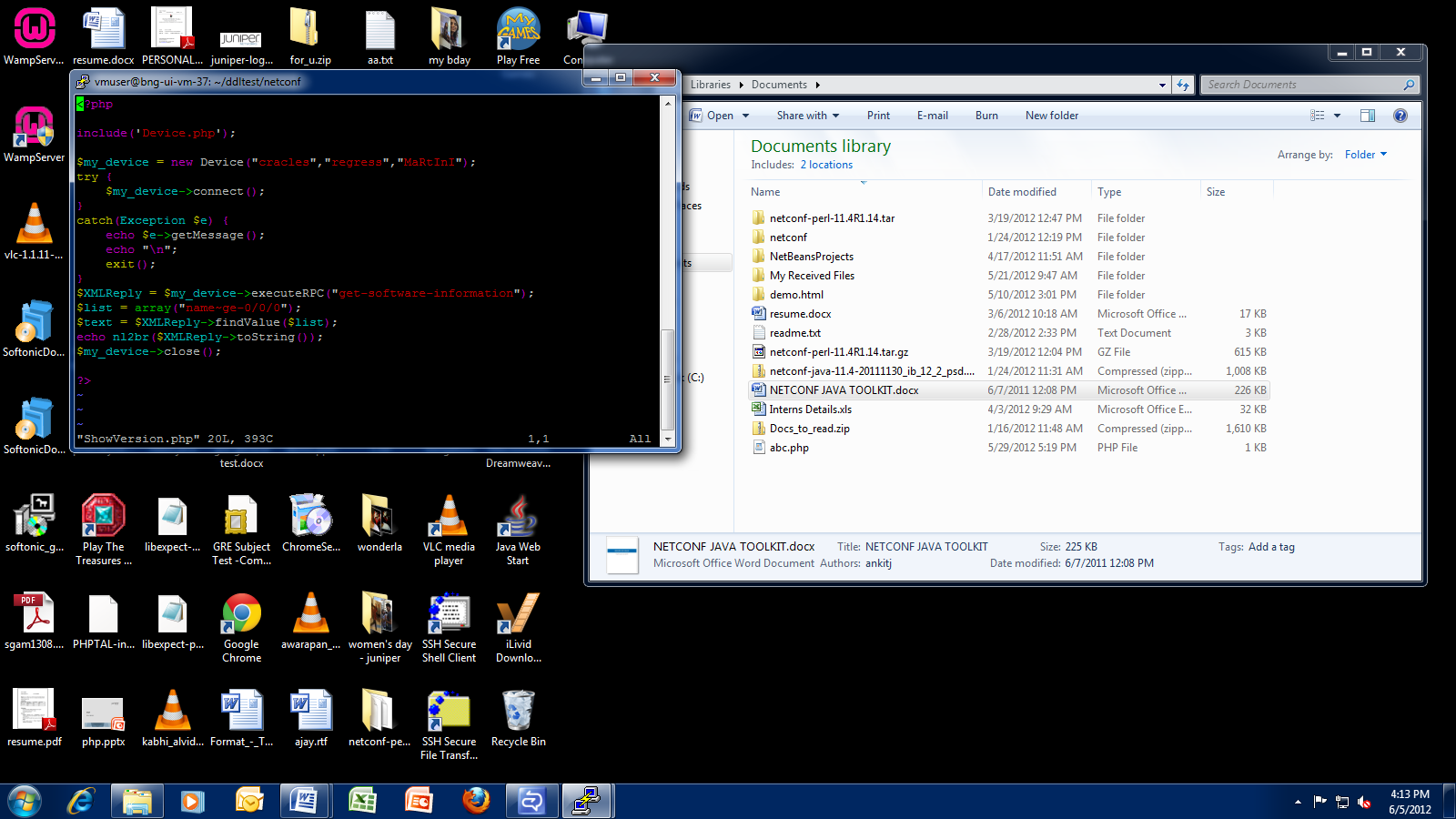
*echo $XMLReply->toString();*

1. **Close the Device.**

*//Code:*

*$my\_device->close();*

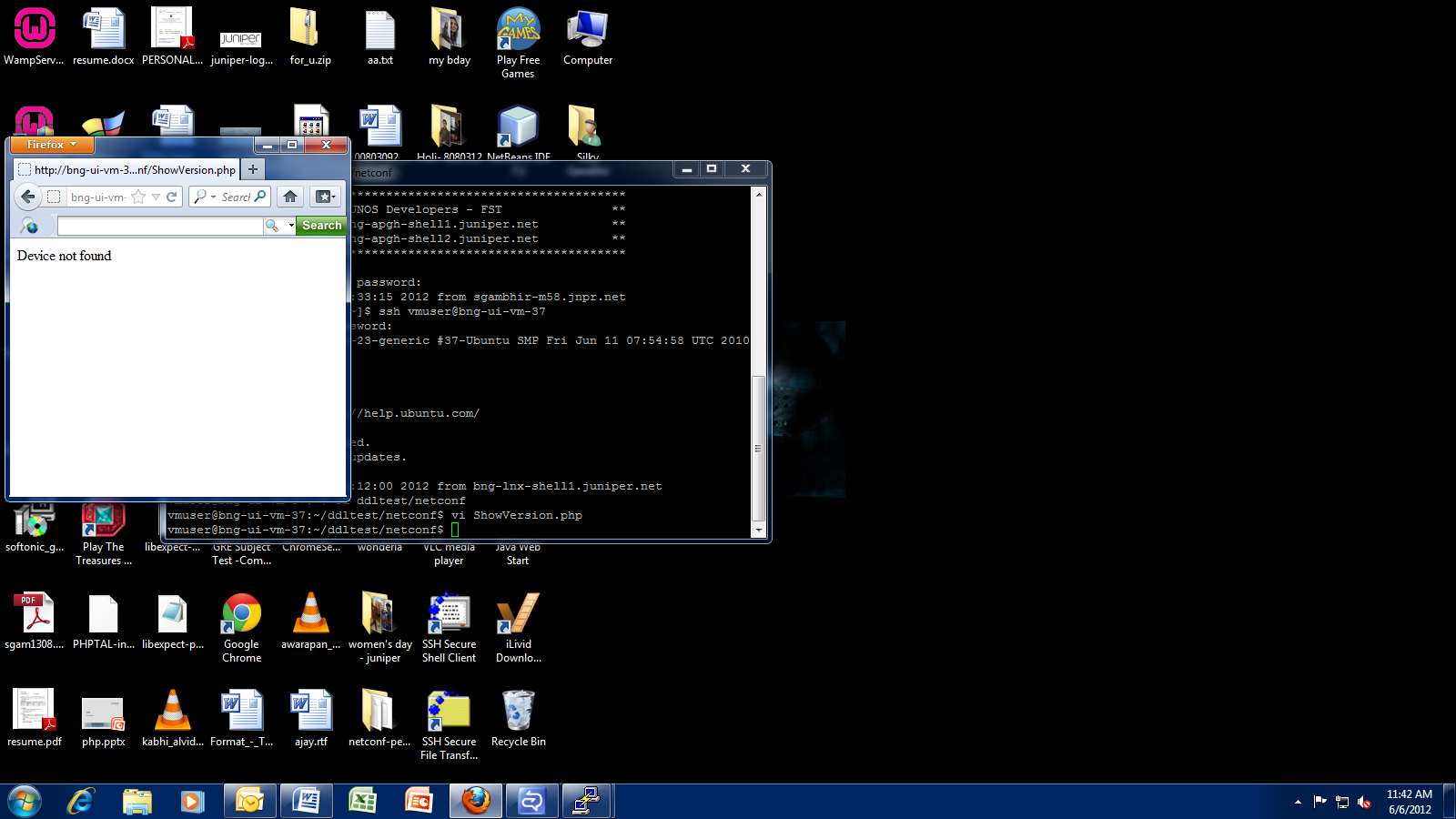
Finally, this is how the ShowVersion.php looks like:

**

*Figure 2.3: Contents of ‘ShowVersion.php’*

URL of the file to be executed

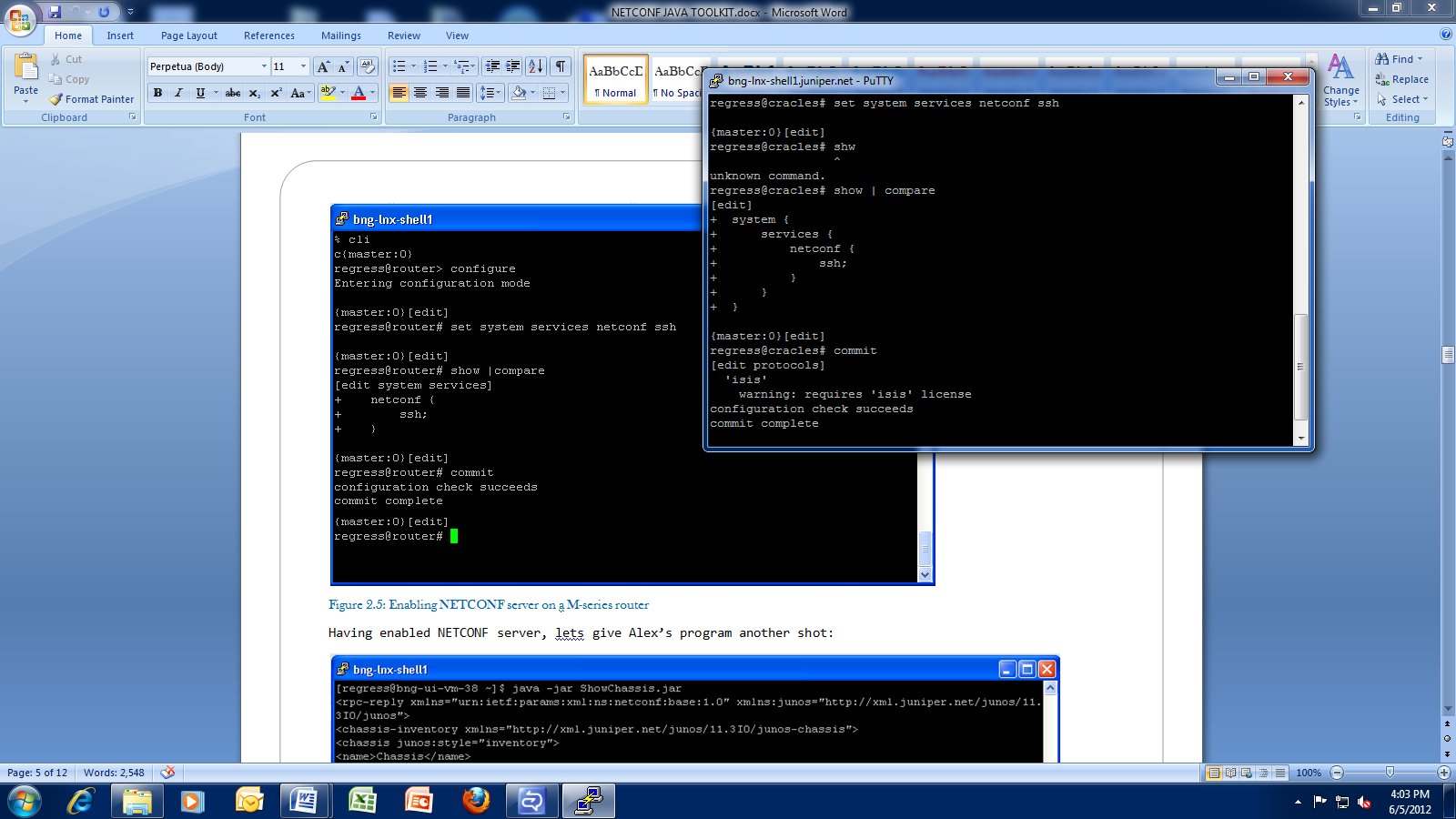
We’ll compile and run Alex code:



*Figure 2.4: Unsuccessful run of ShowVersion*

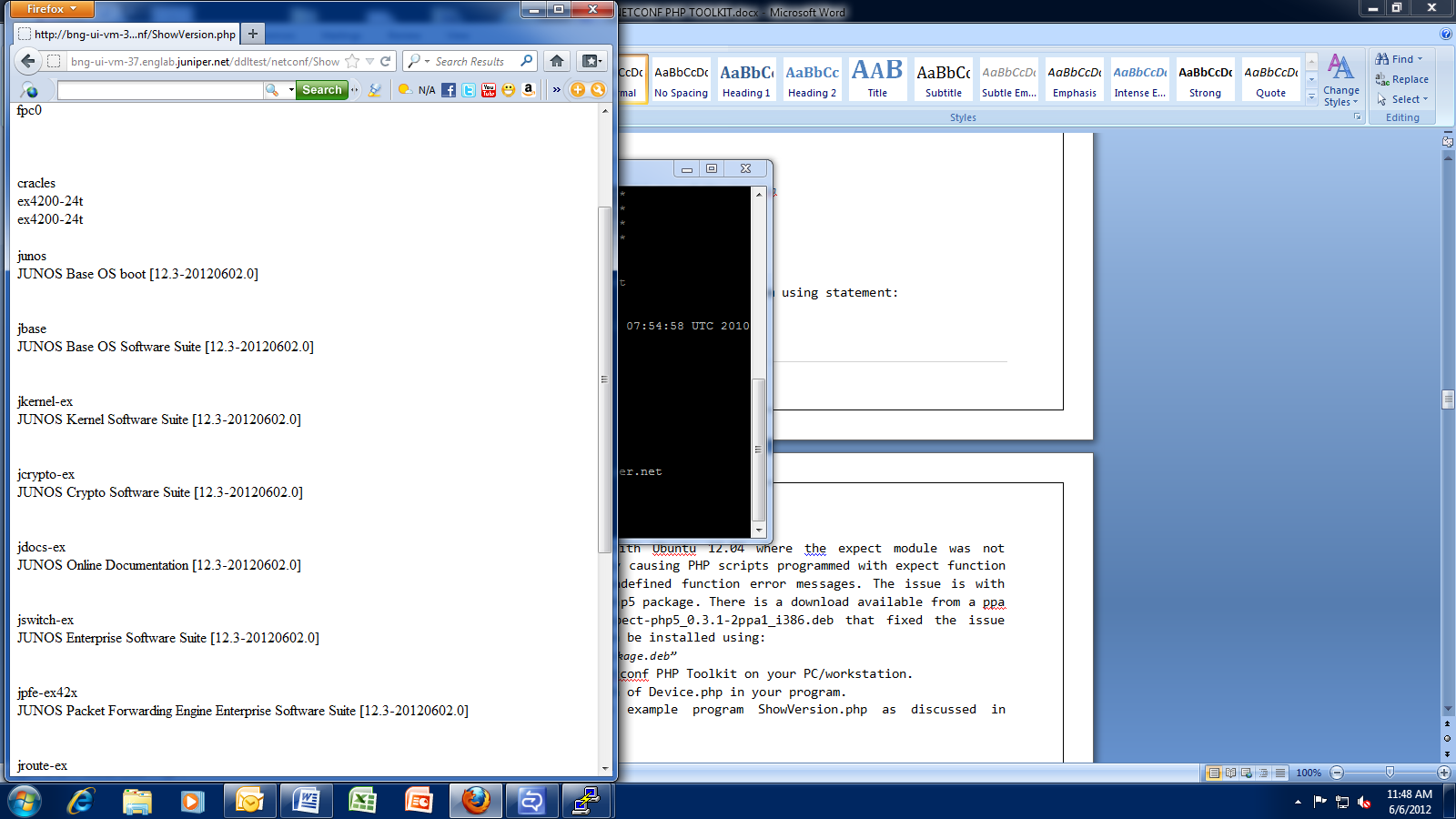
Oops! Device not found error! Turns out that Alex forgot to enable NETCONF server on the JUNOS device.

Alex should setup the NETCONF server on the device first, and ensure SSH-2 connectivity:



*Figure 2.5: Enabling NETCONF server*

Having enabled NETCONF server, lets give Alex’s program another shot:



*Figure 2.5: Successful run of ShowVersion*

1. Installation Instructions

LINUX:

* Install the LAMP on the system.
* Install libexpect-php5 on your system using statement:

*“sudo apt-get install libexpect-php5”*

* Had an issue with Ubuntu 12.04 where the expect module was not loading, thereby causing PHP scripts programmed with expect function to fail with undefined function error messages. The issue is with the libexpect.php5 package. There is a download available from a ppa source - libexpect-php5\_0.3.1-2ppa1\_i386.deb that fixed the issue for me. This can be installed using:

*“sudo dpkg -i package.deb”*

* Download the Netconf PHP Toolkit on your PC/workstation.
* Include the path of Device.php in your program.
* Write and run example program ShowVersion.php as discussed in Section2.

1. What is?

In this section, we’ll look at the basic building blocks of a typical NetconfJava application. We’ll learn what is, and why we need a:

4.1 Device

4.2 XMLbuilder

4.3 XML

* 1. A Device

A netconf.Device object represents an SSH-2 connection.

Typically, one:

1. Creates a Device object.
2. Performs NETCONF operations on the Device object.
3. Finally, one must close the Device and release resources with the close() method.

While creating a Device object, we need to provide the IP address/ DNS name and the authentication details to create an SSH-2 connection to the JUNOS device. This can be a ‘user-password’ based authentication or a RSA/DSA key-based authentication.

We have the option of departing from the default values and specifying the port number for SSH-2 connection and the client capabilities to be sent to the NETCONF server.

* 1. An XML Builder

The NETCONF way of communicating with a JUNOS Device, involves dealing with RPC’s and RPC-Replies, which are XML-encoded data.

The netconf.XMLBuilder and netconf.XML objects help us to create and parse XML-encoded data easily. For details on XML object, refer next section 4.3.

The XMLBuilder object helps us to create new XML object.

* 1. An XML

A netconf.XML object helps us to modify and parse XML-encoded data easily. The user needs to create a XMLBuilder object first, using which he can build a XML object. This makes it easy and convenient to create XML-based RPC’s, configurations and parse the XML-based reply received from the NETCONF server.

The XML object internally maintains a DOMDocument object, corresponding to the XML data it represents.

1. How to?

Having got a brief idea about what is a Device, XMLBuilder and XML, we’ll now learn how to work with each of these objects.

* 1. Create a Device

A Device object can be created using a variety of options:

1. $my\_device = new Device(**$**hostname, $username, $password);

This will create a Device object, which represents an authenticated SSH-2 connection to the mentioned hostname, and a default NETCONF session. The user can now call NETCONF operation methods on this object, most of which return the RPC reply from the NETCONF server.

When the device object is created, both the parties send their hello capabilities. The toolkit looks for “]]>]]>” characters and then sends its hello capabilities to the NETCONF server.

Summary of arguments:

$hostname -> The IP address or the DNS name of the JUNOS device, we’re trying to connect to, via SSH-2. e.g. “10.209.1.23”.

$username -> The username.

$password -> The password required, for user-password based authentication OR key-based authentication. If no password is

required for key-based authentication, pass this argument as

null.

1. $my\_device = new Device(**$**hostname, $username, $password, $port);

This way of creating Device object is used when the SSH-2 connection is on a port number other than the default number, 830.

Summary of new arguments:

$port -> The port number on which to establish the SSH-2 connection.

1. $my\_device = new Device($hostname, $username, $password,

<array>$capabilities);

This way of creating Device object is used when the client capabilities are other than the default capabilities. The user should create an Array and put each capability as an individual entry in the Array.

The default capabilities are:

*urn:ietf:params:xml:ns:netconf:base:1.0*

*urn:ietf:params:xml:ns:netconf:base:1.0#candidate*

*urn:ietf:params:xml:ns:netconf:base:1.0#confirmed-commit*

*urn:ietf:params:xml:ns:netconf:base:1.0#validate*

*urn:ietf:params:xml:ns:netconf:base:1.0#url?protocol=http,ftp,file*

Summary of new arguments:

$capabilities -> The client capabilities to be communicated to the Netconf Server.

1. $my\_device = new Device($hostname, $username, $password, $port,

<array>$capabilities);

This way of creating Device object is used when the SSH

connection is on a port number other than the default number,

830, and the client capabilities are other than the default

capabilities.

Arguments $port and $capabilities are interchangeable.

Having established a Device object, we can now perform NETCONF operations on the device. For a full list of available NETCONF operations, refer to PHPdocs. An example of NETCONF operations is given below:

*$my\_device = new Device(“10.201.12.34”, “alex”, “PaSsWoRd”);*

*//First way:*

*$rpc\_reply\_1 = my\_device->****executeRPC(“get-chassis-inventory”****);*

*//OR*

*//Second way:*

*$builder = new XMLBuilder();*

*$my\_rpc = builder->createNewRPC(“get-chassis-inventory”);*

*$rpc\_reply\_2 = $my\_device->****executeRPC($my\_rpc)****;*

*echo $rpc\_reply\_1->toString();*

*echo $rpc\_reply\_2->toString();*

*$my\_device->close();*

* 1. Create and build upon an XML

To create new RPC’s, configuration or any XML-based data, it’s a good idea to work with the XML object. Using an XML-object, we can easily add/modify elements and attributes.

To facilitate modification of XML content, the XML object maintains an ‘active’ element. This represents the level of hierarchy exposed for modification. To understand this concept, below is an illustration showing how to create a XML object for a given XML data.

**XML to be generated:**

*<configuration>*

*<security>*

*<policies>*

*<policy>*

*<from-zone-name>trust</from-zone-name>*

*<to-zone-name>untrust</to-zone-name>*

*<policy>*

*<name>my-sec-policy</name>*

*<match>*

*<application>junos-ftp</application>*

*<application>junos-ntp</application>*

*<application>junos-ssh</application>*

*<source-address>any</source-address>*

*<destination-address>any</destination-address>*

*</match>*

*<then>*

*<permit/>*

*</then>*

*</policy>*

*</policy>*

*</policies>*

*</security>*

*</configuration>*

|  |  |
| --- | --- |
| **CODE** | **XML Generated** |
| ***$builder*** *= new* ***XMLBuilder****();*  *// Create 3-level hierarchy*  *$****policy*** *= $builder->****createNewConfig****("security","policies","policy");* | *<configuration>*  *<security>*  *<policies>*  *<policy>*  *$****policy 🡪*** |
| *// Append nodes at the 'policy' level*  ***$policy****->****append****("from-zone-name","trust");*  ***$policy****->****append****("to-zone-name","untrust");* | *<configuration>*  *<security>*  *<policies>*  *<policy>*  *<from-zone-name>trust</from-zone-name>*  *<to-zone-name>untrust</to-zone-name>*  ***$policy 🡪*** |
| *// Create the next hierarchy*  ***$policyOne = $policy****->****append****("policy");*  ***$policyOne****->****append****("name","my-sec-policy");* | *<configuration>*  *<security>*  *<policies>*  *<policy>*  *<from-zone-name>trust</from-zone-name>*  *<to-zone-name>untrust</to-zone-name>*  *<policy>*  *<name>my-sec-policy</name>*  ***$policyOne 🡪*** |
| *// Use applications array to append three nodes with same node name*  ***$applications*** *=* ***array****("junos-ftp","junos-ntp","junos-ssh");*  ***$match*** *=* ***$policyOne****->****append****("match");*  ***$match****->****append****("application",$applications);* | *<configuration>*  *<security>*  *<policies>*  *<policy>*  *<from-zone-name>trust</from-zone-name>*  *<to-zone-name>untrust</to-zone-name>*  *<policy>*  *<name>my-sec-policy</name>*  *<match>*  *<application>junos-ftp</application>*  *<application>junos-ntp</application>*  *<application>junos-ssh</application>*  ***$match🡪*** |
| *// Add some more elements under 'match'*  ***$match****->****append****("source-address","any");*  ***$match****->****append****("destination-address","any");* | *<configuration>*  *<security>*  *<policies>*  *<policy>*  *<from-zone-name>trust</from-zone-name>*  *<to-zone-name>untrust</to-zone-name>*  *<policy>*  *<name>my-sec-policy</name>*  *<match>*  *<application>junos-ftp</application>*  *<application>junos-ntp</application>*  *<application>junos-ssh</application>*  *<source-address>any</source-address>*  *<destination-address>any</destination-address>*  ***$match 🡪*** |
| *// Add 'then', 'permit'*  ***$policyOne****->****append****("then")->****append****("permit");* | *<configuration>*  *<security>*  *<policies>*  *<policy>*  *<from-zone-name>trust</from-zone-name>*  *<to-zone-name>untrust</to-zone-name>*  *<policy>*  *<name>my-sec-policy</name>*  *<match>*  *<application>junos-ftp</application>*  *<application>junos-ntp</application>*  *<application>junos-ssh</application>*  *<source-address>any</source-address>*  *<destination-address>any</destination-address>*  *</match>*  *<then>*  *<permit/>*  *</then>*  ***$policyOne 🡪*** |
| *//That’s all* | *<configuration>*  *<security>*  *<policies>*  *<policy>*  *<from-zone-name>trust</from-zone-name>*  *<to-zone-name>untrust</to-zone-name>*  *<policy>*  *<name>my-sec-policy</name>*  *<match>*  *<application>junos-ftp</application>*  *<application>junos-ntp</application>*  *<application>junos-ssh</application>*  *<source-address>any</source-address>*  *<destination-address>any</destination-address>*  *</match>*  *<then>*  *<permit/>*  *</then>*  *</policy>*  *</policy>*  *</policies>*  *</security>*  *</configuration>* |

Table 1

* 1. Parse an RPC Reply

Post a NETCONF operation, an RPC reply is received from the Device object in form of an XML object.

*$rpc\_reply = $my\_device->executeRPC(“get-software-information”);*

There are two approaches to parse an XML Reply:

1. Fetch the Document object and use the native parsing methods available in standard PHP library for Document object. This method is useful for the flexibility and options available in terms of the standard PHP library methods.

*$doc = $rpc\_reply->getOwnerDocument();*

*//Now use standard PHP library methods to handle the Document object.*

1. Use the inbuilt findValue($list) method on XML object.
   1. Using findValue($list) method:

We can use this method to quickly find out the **text value of a given element** at any level of hierarch.

e.g. for the given RPC-Reply format:

*<rpc-reply>*

*<interface-information>*

*<physical-interface>*

*<name>ge-0/0/0</name>*

*<admin-status>up</admin-status>*

*……*

Lets say we want to find out the admin-status of the physical interface named ‘ge-0/0/0’. Being aware of the RPC-Reply format, we can get this information instantly:

*$rpc\_reply = $my\_device->executeRPC(“get-software-information”);*

*$list = array("interface-information","physical-interface","name****~****ge-0/0/0", “admin-status");*

*//NOTE THE ‘~’ CHARACTER IN ‘name~ge-0/0/0’ TO IDENTIFY THE PARTICULAR ELEMENT.*

*$admin\_status = rpc\_reply.findValue(list);*

*echo $admin\_status;*

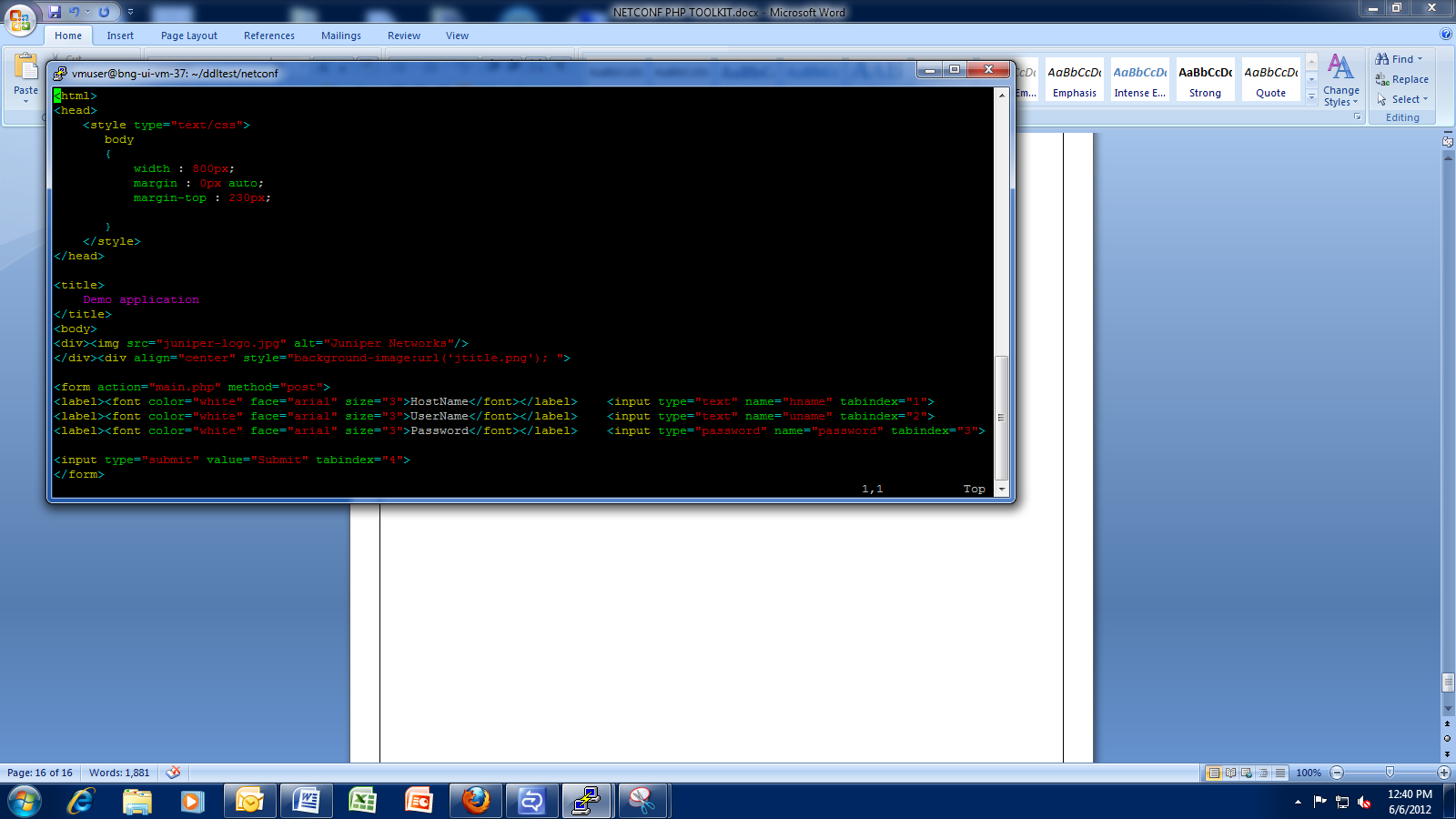
*$admin\_status now returns “up”*

1. Lets learn with examples

Having ‘somewhat’ understood how to use the NETCONF PHP API’s, lets explore some small applications. The code is presented in sections below.

* 1. Web-based CLI(only for operational mode commands)

1. Create a web-page that will ask for username, hostname and password.



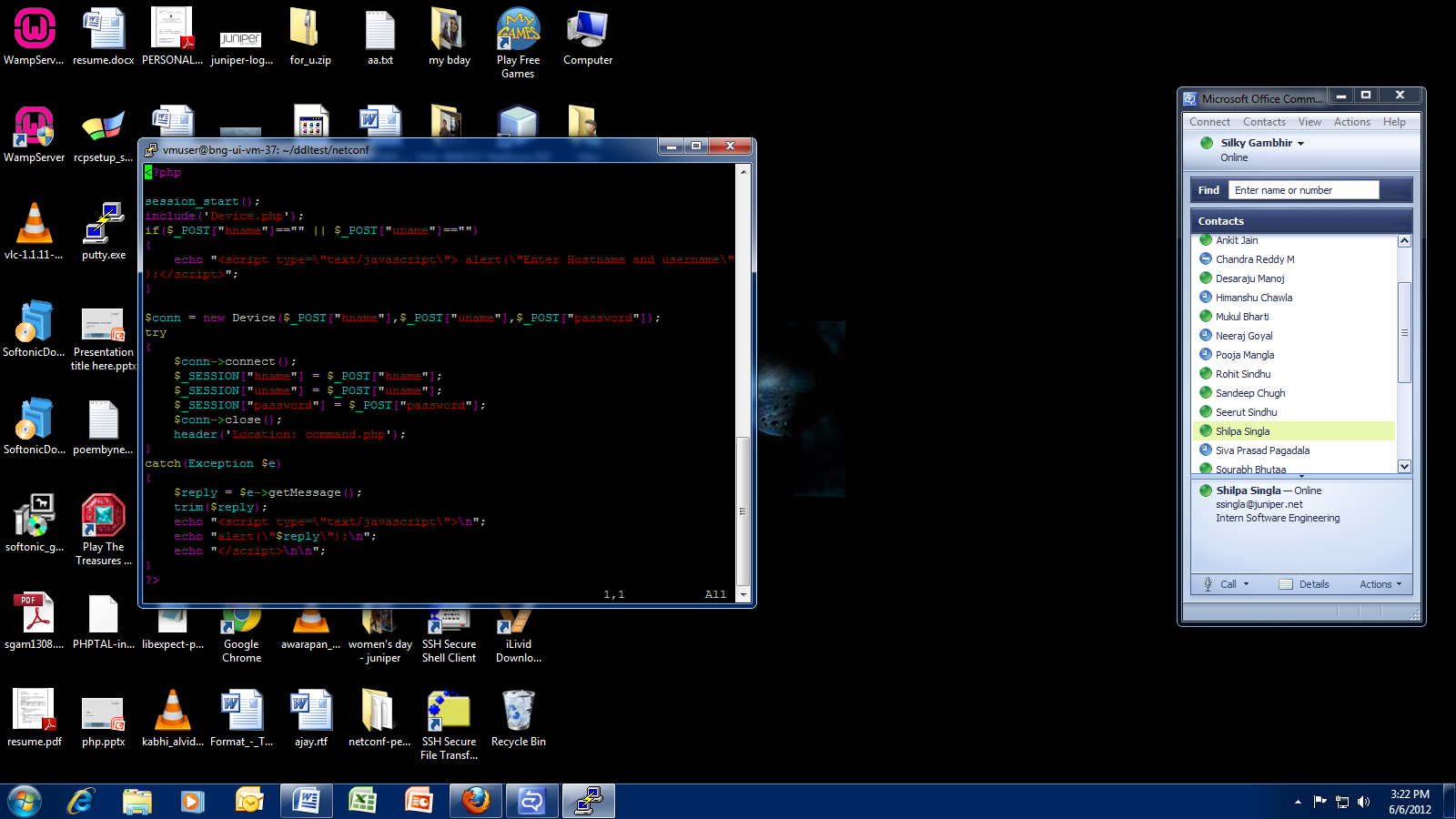
*Figure 6.1: Contents of “login.html”*

It’ll look like this:



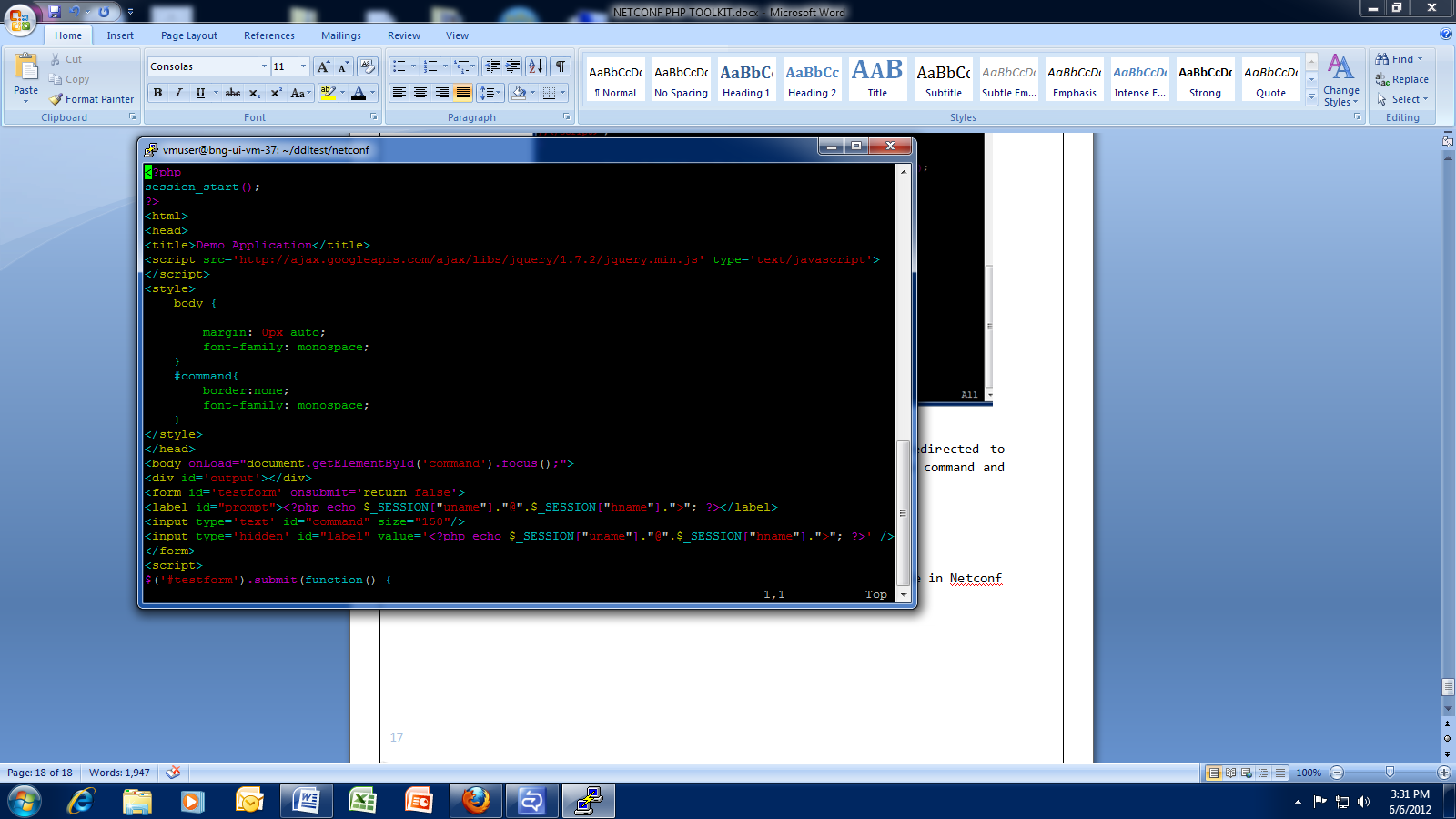
*Figure 6.2: login.html*

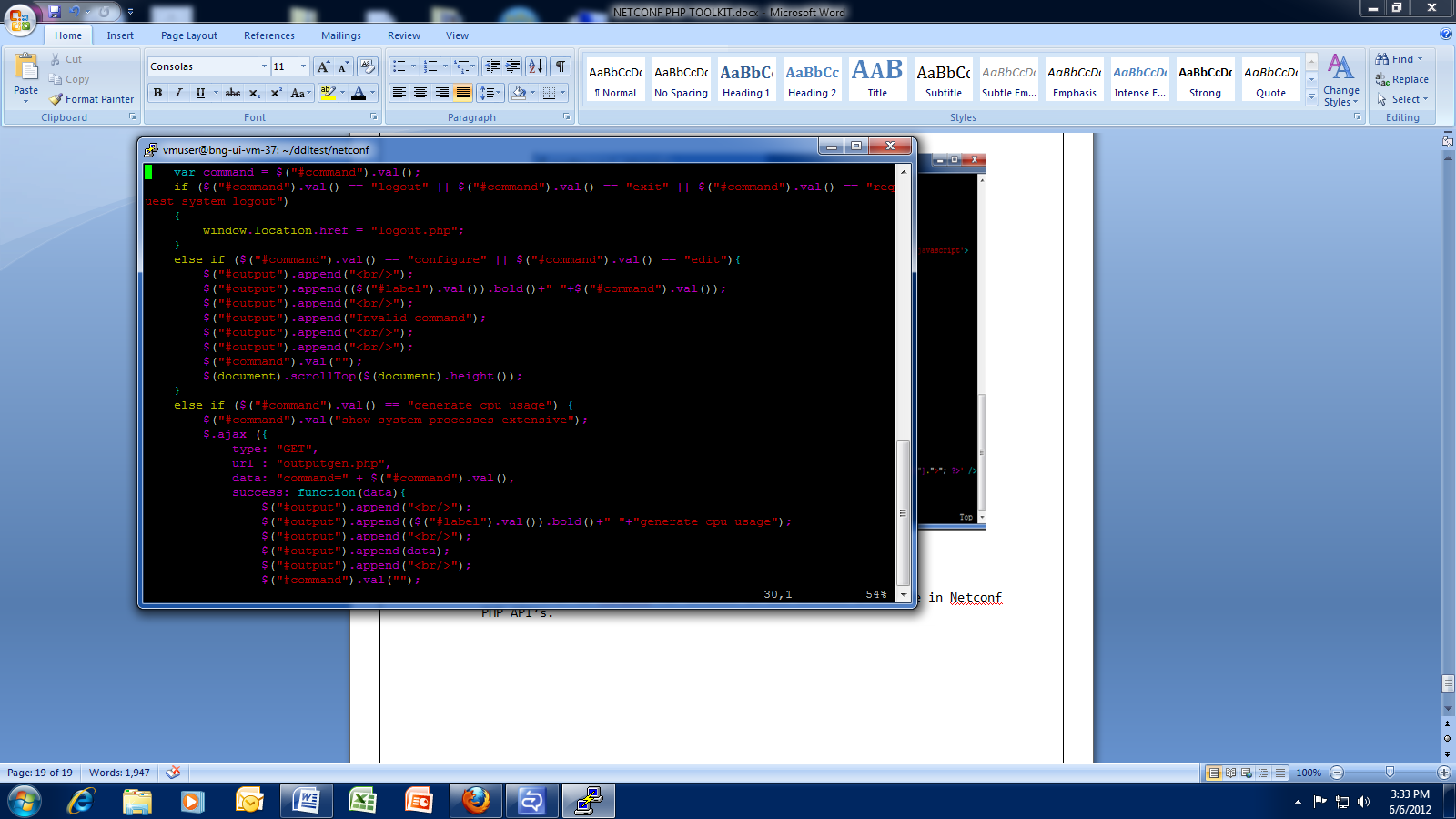
1. When user clicks on “Submit” button then the values will be sent to “login.php” in which the connection is made to the Device.

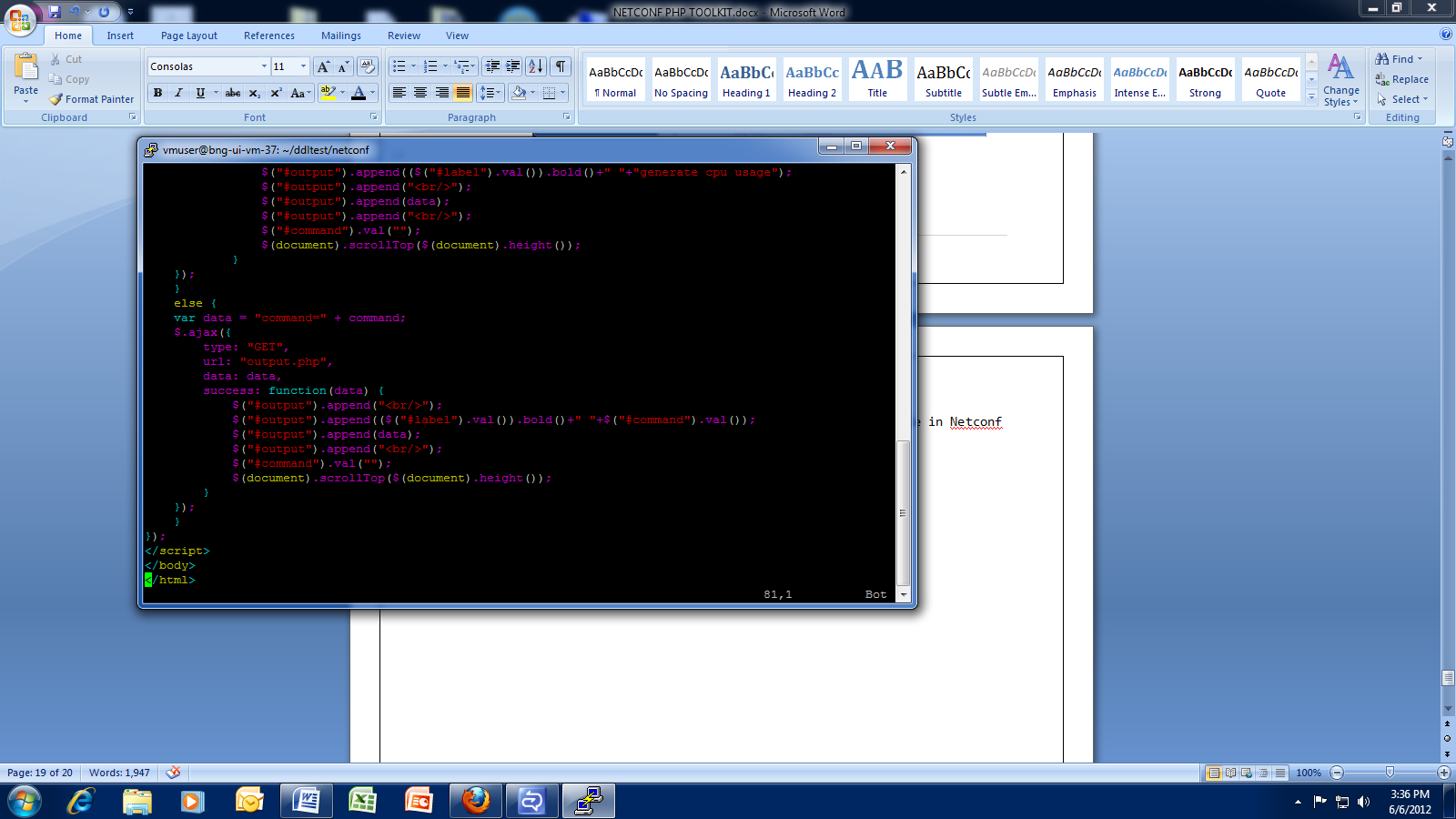


*Figure 6.3: Contents of “login.php”*

1. After connecting to the Device, the control is redirected to “input.php”, where you can give any operational mode command and can see the output.

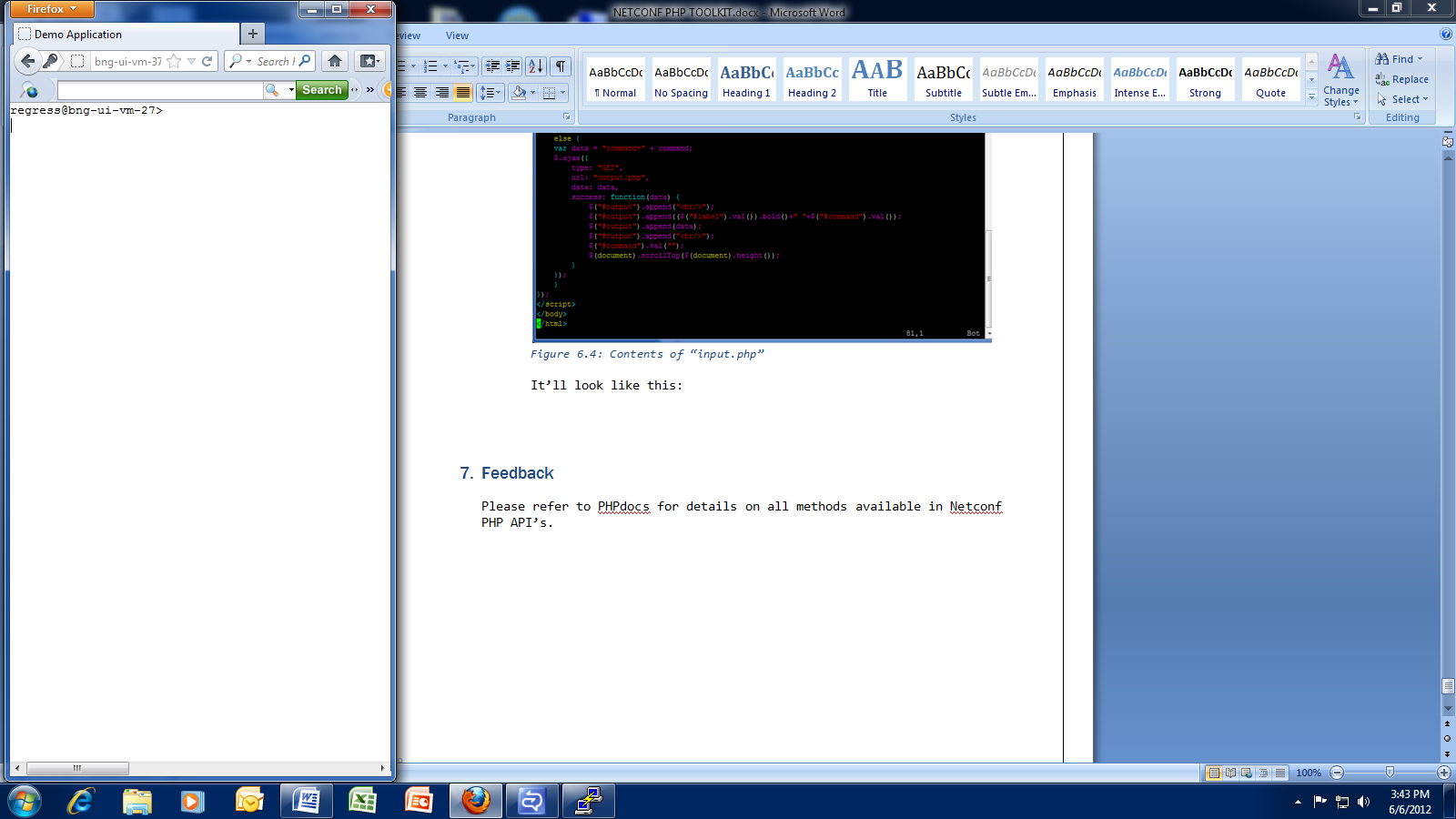






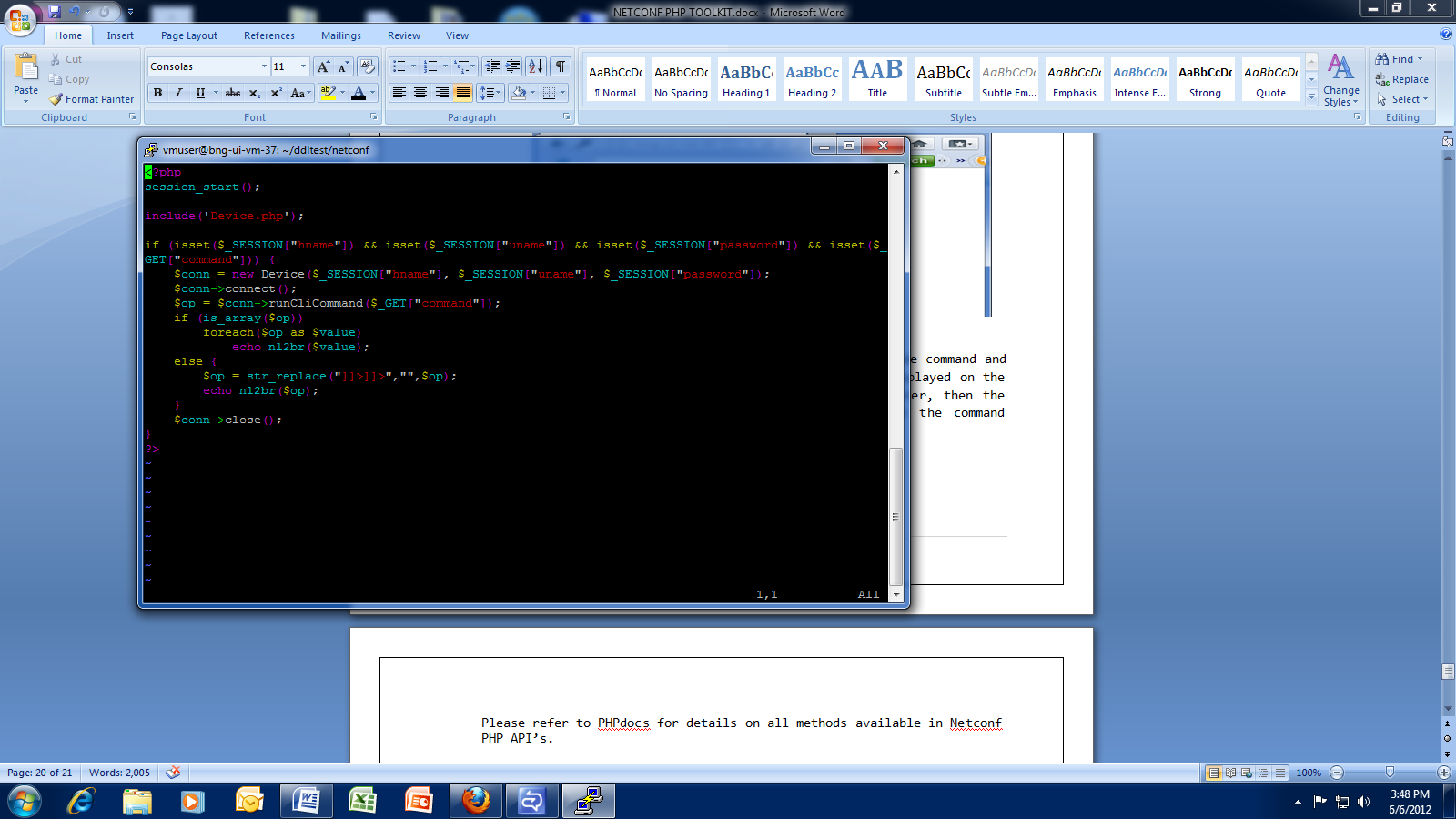
*Figure 6.4: Contents of “input.php”*

It’ll look like this:



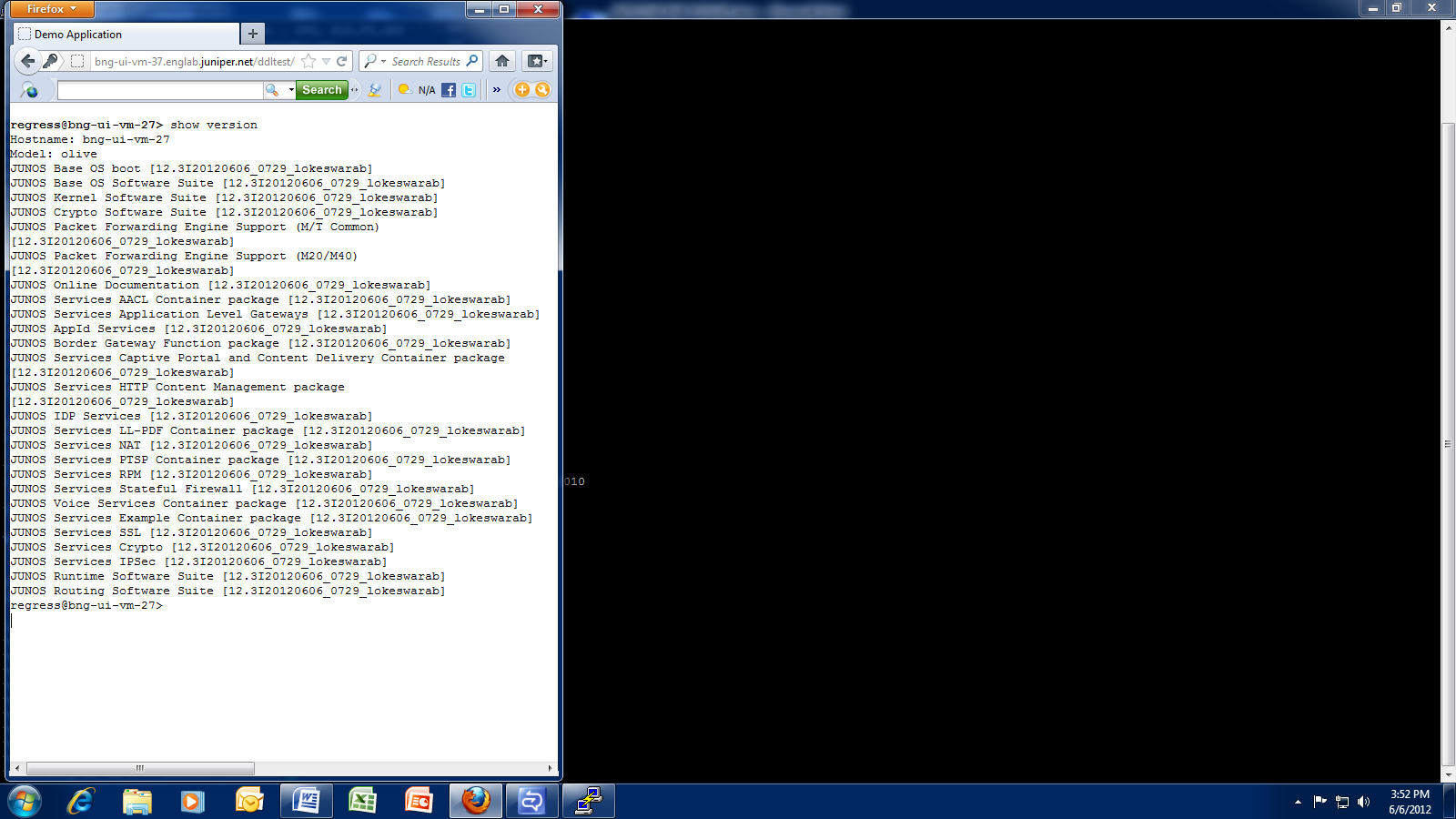
*Figure 6.5: input.php*

1. Now at the prompt, you can type any operational mode command and then press enter. The resultant output will be displayed on the screen. When the user types command and press enter, then the control will be directed to “output.php” to get the command executed.



*Figure 6.6: Contents of “output.php”*

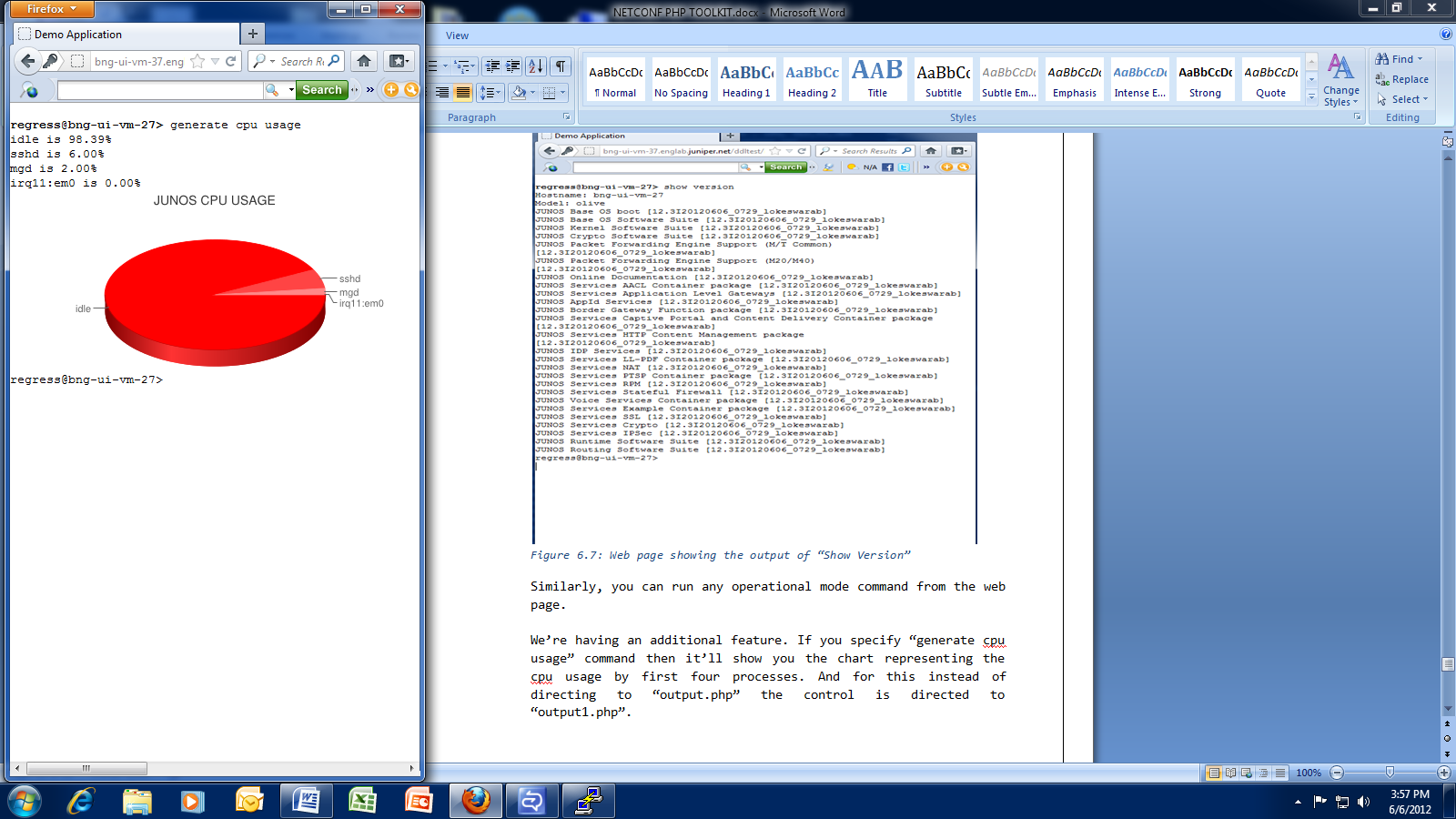
Lets give a try for Show Version command:



*Figure 6.7: Web page showing the output of “Show Version”*

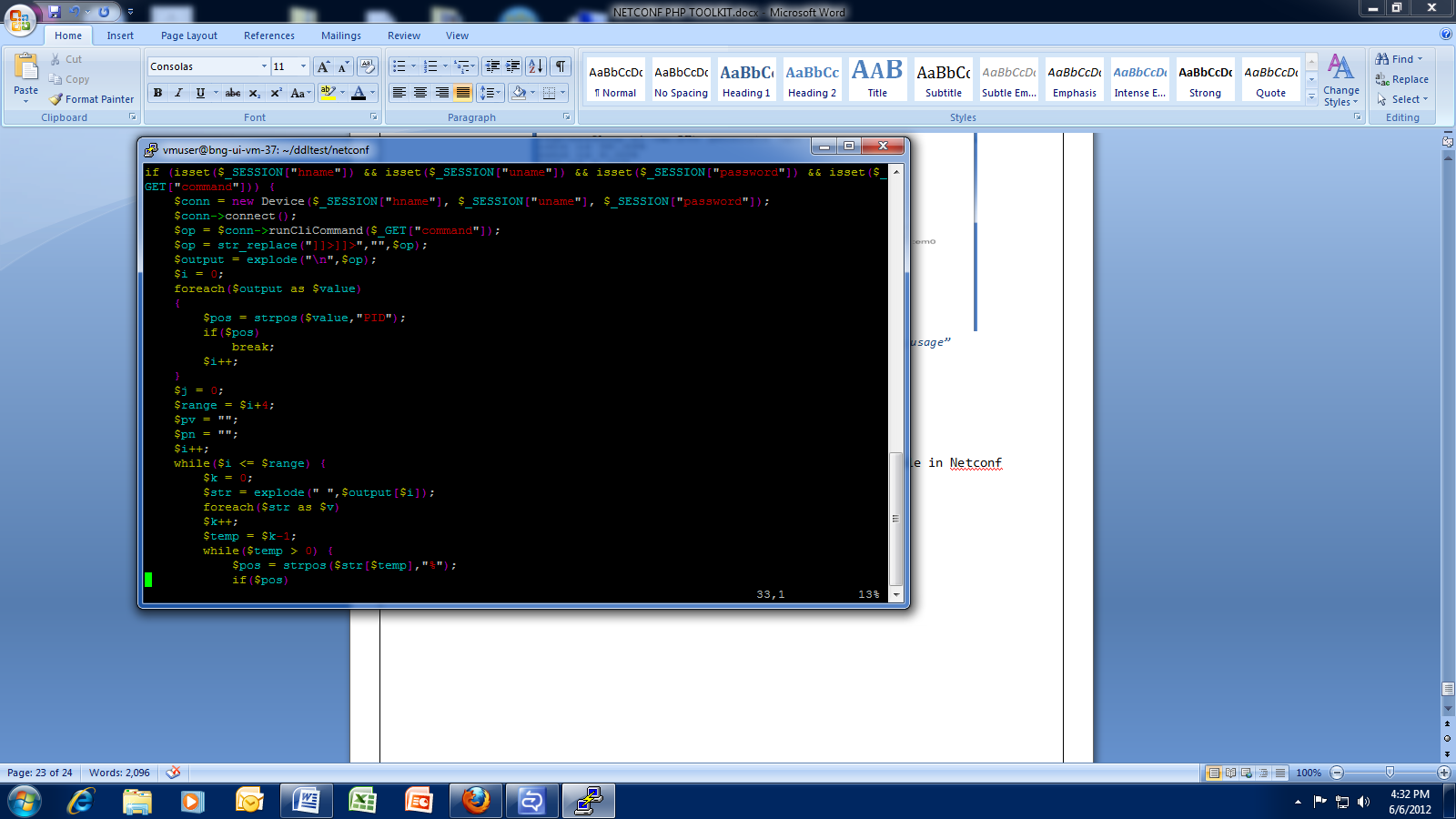
Similarly, you can run any operational mode command from the web page.

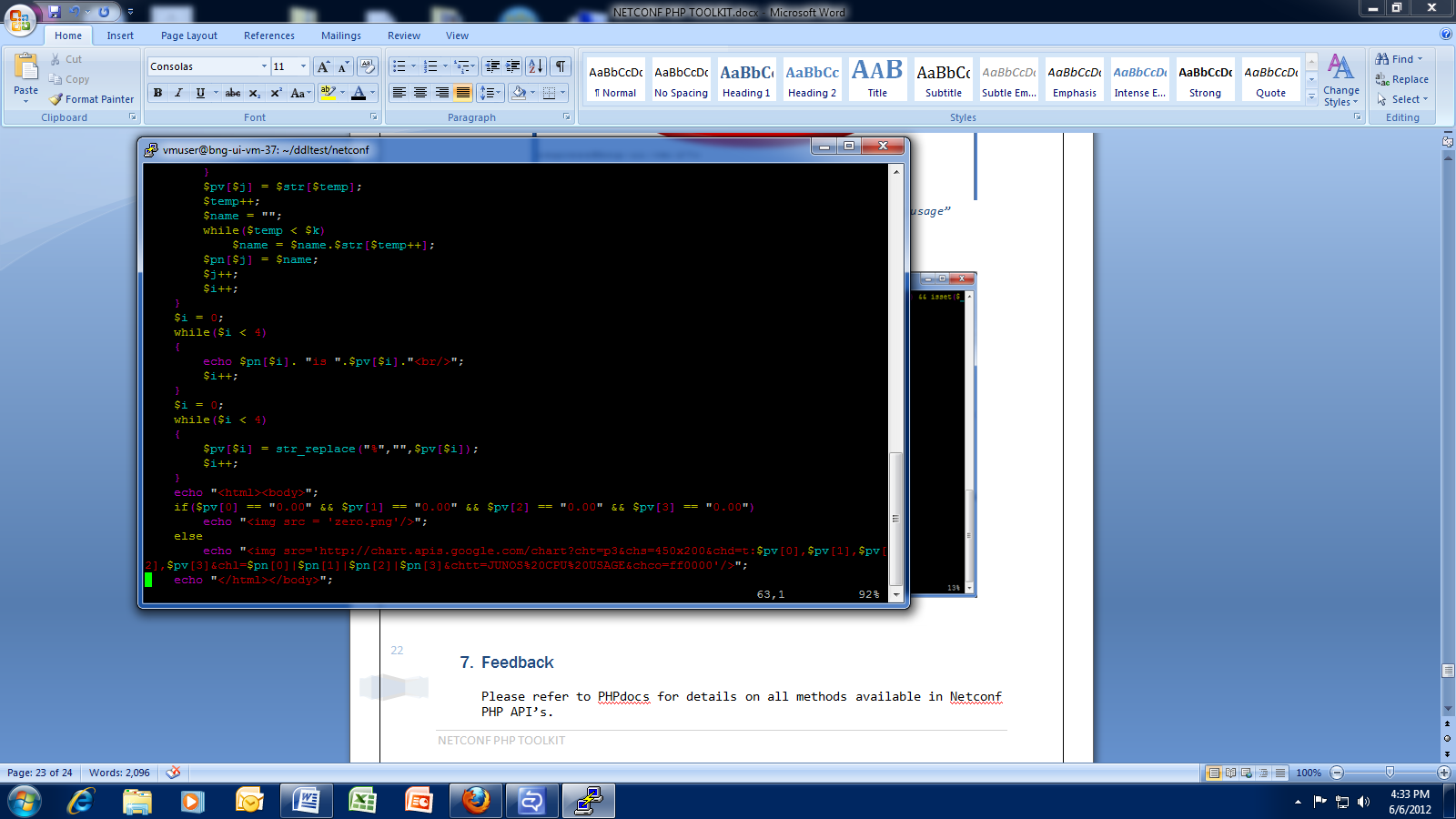
We’re having an additional feature. If you specify “generate cpu usage” command then it’ll show you the chart representing the cpu usage by first four processes. And for this instead of directing to “output.php” the control is directed to “outputgen.php”.



*Figure 6.8: Web page showing the output of “generate cpu usage”*

And the contents of outputgen.php:





*Figure 6.9: Contents of “outputgen.php”*

1. Feedback

Please refer to PHPdocs for details on all methods available in Netconf PHP API’s.