Chapter 10 – Defensive Coding

This is likely one of those chapters that you may be tempted to skip over and possibly ignore, as it seems a little light on coding detail and heavy on high-level concepts. If you choose to skip it, make sure you return at some point in the future and find out how you could have saved yourself a whole lot of pain and time. ;-)

This chapter looks at several techniques and practices to make your code more robust and to help you figure out what happened when things inevitably break or do not provide the results you expected. We'll look at various coding and style tips to ensure your scripts run as painlessly as possible, are easier to debug, and at least fail in a way that indicates what went wrong.

Not all of the suggested tips in this chapter must be applied to your scripts. But as you write longer, more complex scripts, you will definitely want to adopt some of the guidelines provided here.

The guidelines provided in this chapter will cover the following subjects:

- Documenting your code
- Naming conventions
- Data type checking
- Error trapping
- Code debugging
- Profiling your code

To demonstrate these areas, we'll look at a "poorly written" script and create a new, improved version as we explore each area listed above.

"Bad Script"

Here is our sample "bad script". It's syntactically correct but needs some improvement. We'll improve is as we look at each subject area in this chapter:

```
# filename: ch10-01-bad-script.rsc

:local w {"mikrotik.com"; "www.google.com"; "twitter.com"};

:foreach s in=$w do={
    :local i [:resolve $s];
    :local p [/ping $i count=3 ];
    :local u [/tool fetch url=("https://$s") mode=https http-method=get \
        as-value keep-result=no];

    :put "==========================================";
    :put ("Site: $s");
    :put ("IP Address = $i");
    :put ("Ping success = $p/3");
    :put ("Page d/load duration: " . $u->"duration");
    :put "===========================================";

}
```

As the script is relatively simple, you'll probably be able to work out what it does just by reading through the code. If not, you can copy and paste it into a script file on your MikroTik device and see what happens. Its operation will become apparent as you work through this chapter. Although the code is simple, it has many pitfalls, which we'll explore in this chapter by optimizing the script to make it more production ready.

Documenting Your Code

It's almost inevitable that at some stage in the future, you (or one of your colleagues) will return to a script you've written to possibly reuse or adjust some of your code. While the logic of your code is self-evident when you're creating it, a few weeks or months later, you'll likely find you don't remember the finer details of the code and the rationale for specific coding decisions.

As discussed previously, RouterOS has a built-in commenting feature that allows the insertion of code comments which can be used to document your code. By inserting comments directly within your code, anyone reviewing your code will more easily understand the *intent* of your code and the reasons behind specific coding decisions.

Comments are inserted by starting a sequence of text with the hash (#) symbol and end at a line-end. If you wish to have a multi-line block of commented text, each line in the block must start with a hash; comments cannot span multiple lines.

Let's look at how we can improve our script with commenting:

```
# filename: ch10-02-bad-script.rsc
#
# A simple script to perform a series of tests on a
# list of web sites.

# define list of websites to test
:local w {"mikrotik.com"; "www.google.com"; "twitter.com"};

# step through each website and perform various tests
:foreach s in=$w do={

    # try a DNS resolution of this site
    :local i [:resolve $s];

    # try pinging the IP address of this site
    :local p [/ping $i count=3 ];

    # try fetching the web page of this site
    :local u [/tool fetch url=("https://$s") mode=https http-method=get \
        as-value keep-result=no];

    # print out a summary report for this site
    :put "==========================================";
    :put ("Site: $s");
    :put ("IP Address = $i");
    :put ("Ping success = $p/3");
    :put ("Page d/load duration: " . $u->"duration");
    :put "===========================================";
}
```

As you can see above, the script's intent is far more apparent as we added comments throughout the script.

At the beginning of the script, a simple description outlines the intended purpose of the script. This is followed by a series of inline comments describing each stage of the script's operation. As well as being a helpful reminder of what each code block does, comments can be beneficial when debugging code when the intent of the code is clearly spelt out. The intent may not match the actual code that is included due to coding errors. Understanding what the code is *supposed* to do can be very useful when figuring out coding problems.

Naming Conventions

Although we've certainly improved things in our sample script, reading through the code, it's not immediately obvious what each variable represents. The code, as it stands, uses single-letter variable names such as "`i`", "`p`", and "`u`".

When I was coding the script, in my mind, it was evident that these represent an "IP address", "ping result", and a "URL". I just took the first letter of each word and used them as a variable name. Whether I would still remember what they represent a few months later is debatable. After refreshing my memory by reading through the code, it may jog my memory. But if I asked a colleague to review the code, they may not find it as easy to understand what the single-letter variables represent.

We should use well-named variables in our code to make things easier and avoid coding errors. We should also use a convention that makes variables easy to spot in our code. The convention I've suggested previously in this book is to use variables that begin with capitalised letters to make them easily distinguishable from RouterOS commands and keywords, which all begin with lowercase letters. If multiple words are included in a variable name, each word should also begin with a capital letter. (*Please refer to the "Variable Naming Convention" section of chapter 6 for more in-depth coverage of variable naming conventions*).

In our sample script, we'll update all variable names to be easily identifiable and use names that indicate their purpose:

```
# filename: ch10-03-bad-script.rsc
#
# A simple script to perform a series of tests on a
# list of web sites.

# define list of websites to test
:local WebSites {"mikrotik.com"; "www.google.com"; "twitter.com"};

# step through each website and perform various tests
:foreach SiteName in=$WebSites do={

    # try a DNS resolution of this site
    :local IpAddress [:resolve $SiteName];

    # try pinging the IP address of this site
    :local PingResult [/ping $IpAddress count=3 ];

    # try getting the web page of this site
    :local UrlFetch [/tool fetch url=("https://$SiteName") mode=https \
        http-method=get as-value keep-result=no];

    # print out a summary report for this site
    :put "========================================";
    :put ("Site: $SiteName");
    :put ("IP Address = $IpAddress");
    :put ("Ping success = $PingResult/3");
    :put ("Page d/load duration: " . $UrlFetch->"duration");
    :put "========================================";
}
```

When reviewing the code with the improved variable names, it's more obvious what each variable is used for.

In addition to the variable naming convention described above, I strongly recommend a couple of other variable naming conventions for functions and constant values:

Function Names

Although we covered functions in chapter 9, it's worth re-iterating the advice it provided. Functions are essentially variables that have a block of code assigned to them instead of a data value. To differentiate them from "standard" variables in your code, I strongly recommend you add a "`Func`" suffix to the variable name. This makes them far easier to spot when defined or used within your code. Please refer to chapter 9 for function naming, definition, and usage examples.

Constants

Sometimes, we may wish to define values that we do not intend to be changed during the lifetime of a script. For instance, we may define a filename we'd like to dump data into. The filename should not change during the course of the script; we expect it to remain constant. In many programming languages, these values are assigned to "constants". Constants are given values in a similar way to variables, but once set, their value remains unalterable.

RouterOS does not support the concept of constants, so we have to get creative and re-purpose variables to indicate that we wish their data to be treated as a constant. It's important to identify variables that we wish to be treated as constants in our code so that we do not update their value during the flow of a script. To identify variables to be treated as constants, I suggest fully capitalising their name. This explicitly calls them out as something that should not be updated during the course of your script. (*Please refer to the "Variable Naming Convention" section of chapter 6 for more in-depth coverage of variable naming conventions*).

Here is a quick example of how we might define a constant value:

```
# demo of using a constant
:local MBPS 1000;
:local KbpsValues { 2000; 10000; 5000 };

foreach Value in=$KbpsValues do={
    :put ("$Value kbps is equivalent to: " . ($Value/$MBPS) . " mbps");
}
```

In this example, we convert values expressed in kilobits per second to megabits per second. The divider we use to convert kbps to mbps should always remain as "1000"; therefore, we declare it as the constant value "MBPS".

Note that there is nothing special about variables declared this way; their data may still be changed in your code. Using capitalisation is just a visual hint to yourself or others