



REVISION PAGE

Revision	Date	Description
Revision 0.0	December 2019	Original

Table of Contents

PREFACE	8
2. INTRODUCTION	8
2.1 WHY ERLANG	8
2.2 ERLANG DESIGN GOALS.....	9
2.3 ERLANG PROGRAMMING LANGUAGE	9
3. INSTALLATIONS	10
3.1 INSTALLATION OF ERLANG OTP 22.1	10
3.2 GIT.....	14
3.2.1 Installation of Git 2.24.1.2.....	14
3.3 GNU MAKE FOR WINDOWS	26
3.3.1 Installation of GNU Make for Windows.....	26
3.4 REBAR3	26
3.4.1 Installation of Rebar3.....	27
CHAPTER 1 – STARTING A PROJECT	34
CHAPTER 2 - ERLANG IN PRODUCTION	48
CHAPTER 3 - ERLANG OTP CLUSTERS	83
APPENDIX A ABBREVIATIONS AND ACRONYMS	95
APPENDIX B BIBLIOGRAPY	96

LIST OF FIGURES

FIGURE 1 - OPEN FILE - SECURITY WARNING.....	10
FIGURE 2 - CHOOSE COMPONENTS	11
FIGURE 3 - CHOOSE INSTALL LOCATION.....	12
FIGURE 4 - CHOOSE START MENU FOLDER.....	13
FIGURE 5 - INSTALL/REPAIR MICROSOFT VISUAL C++ REDISTRIBUTABLES (x64).....	13
FIGURE 6 – INSTALLATION COMPLETE	14
FIGURE 7 - EShell V10.5 SESSION.....	14
FIGURE 8 - EShell QUIT SESSION.....	14
FIGURE 9 - GIT 2.24.1.2 SETUP INFORMATION.....	15
FIGURE 10 - GIT 2.24.1.2 SETUP SELECT DESTINATION LOCATION.....	16
FIGURE 11 - GIT 2.24.1.2 SETUP SELECT COMPONENTS	17
FIGURE 12 - GIT 2.24.1.2 SETUP SELECT START MENU FOLDER.....	18
FIGURE 13 - GIT 2.24.1.2 SETUP CHOOSING THE DEFAULT EDITOR USED BY GIT	19
FIGURE 14 - GIT 2.24.1.2 SETUP ADJUSTING YOUR PATH ENVIRONMENT.....	20
FIGURE 15 - GIT 2.24.1.2 SETUP CHOOSING HTTPS TRANSPORT BACKEND.....	21
FIGURE 16 - GIT 2.24.1.2 SETUP CONFIGURING THE LINE ENDING CONVERSIONS	22
FIGURE 17 - GIT 2.24.1.2 SETUP CONFIGURING THE TERMINAL EMULATOR TO USE WITH GIT BASH	23
FIGURE 18 - GIT 2.24.1.2 SETUP CONFIGURING EXTRA OPTIONS.....	24
FIGURE 19 - GIT 2.24.1.2 SETUP CONFIGURING EXPERIMENTAL OPTIONS	25
FIGURE 20 - GIT 2.24.1.2 SETUP COMPLETING THE GIT SETUP WIZARD	26
FIGURE 21 - GIT CLONE OF REBAR3 REPOSITORY.....	27
FIGURE 22 - GIT CLONED OF REBAR3 COMPLETED	27
FIGURE 23 - CHANGE DIRECTORY TO REBAR3 FOLDER	27
FIGURE 24 - REBAR3 BOOTSTRAP COMMAND.....	28
FIGURE 25 - REBAR3 UNLOCK COMMAND	28
FIGURE 26 - REBAR3 UPDATE COMMAND	28
FIGURE 27 - REBAR3 CLEAN --ALL COMMAND.....	29
FIGURE 28 – MKDIR \$USERPROFILE\BIN DIRECTORY AND COPY REBAR3, REBAR3.CMD FILES.....	29
FIGURE 29 - ADD THE %USERPROFILE%\BIN TO USER VARIABLE PATH	30
FIGURE 30 - REBAR3 VERSION COMMAND	30
FIGURE 31 - REBAR_CACHE_DIR AND REBAR_GLOBAL_CONFIG_DIR.....	31
FIGURE 32 - USERS HOME DIRECTORY	31
FIGURE 33 - MKDIR .CONFIG AND BOOTCAMP.....	32
FIGURE 34 - MKDIR REBAR3	32
FIGURE 35 - MKDIR TEMPLATES	32
FIGURE 36 - TEMPLATES LISTING.....	33
FIGURE 37 - CD ~/BOOTCAMP.....	34
FIGURE 38 - REBAR3 NEW RELEASE CHP01.....	34
FIGURE 39 – REBAR3 RELEASE.....	39
FIGURE 40 – CHP01.CMD CONSOLE COMMAND.....	39
FIGURE 41 - CHP01 CONSOLE WINDOW.....	40
FIGURE 42 - REBAR3 RELEASE OUTPUT	41
FIGURE 43 - CHP01.BAT NO ARGUMENTS	42
FIGURE 44 – REBAR3 AS PROD TAR	43
FIGURE 45 – INSTALLATION OF CHP01 PRODUCTION RELEASE VIA TAR –ZXVF CHP01-0.1.0.TAR.GZ	44
FIGURE 46 – BIN/CHP01.CMD INSTALLED AS WINDOWS SERVICE.....	44
FIGURE 47 – BIN/CHP01.CMD START COMMAND.....	45
FIGURE 48 - BIN/CHP01.CMD PING COMMAND.....	45
FIGURE 49 - COPY BASH.BASHRC TO ~/.BASHRC.....	45
FIGURE 50 - BIN/CHP01.CMD ATTACH COMMAND	46
FIGURE 51 - BIN/CHP01.CMD COMMAND.....	46
FIGURE 52 - BIN/CHP01 UNINSTALL COMMAND.....	46
FIGURE 53 - REBAR3 NEW RELEASE CHP02.....	48

FIGURE 54 - CHP02_APP ADD SASL, LAGER CALLS.....	52
FIGURE 55 – ./CHP02.CMD CONSOLE.....	52
FIGURE 56 – LS LOG COMMAND.....	52
FIGURE 57 – CAT LOG/CONSOLE.LOG COMMAND.....	53
FIGURE 58 - CONTENTS OF CHP02_REPORT.CMD COMMAND FILE.....	54
FIGURE 59 – START ./CHP02_REPORT.CMD COMMAND FILE.....	54
FIGURE 60 - CHP02 RB:LIST(). COMMAND.....	55
FIGURE 61 - CHP02 RB:SHOW(1), RB:SHOW(10) AND RB:SHOW(17) COMMANDS.....	56
FIGURE 62 - ADD ELARM MODULE TO STARTUP OF CHP02 APPLICATION.....	59
FIGURE 63 - ./CH02 CMD CONSOLE.....	59
FIGURE 64 - CHP02 OBSERVER SYSTEM TAB.....	59
FIGURE 65 - CHP2 OBSERVER APPLICATIONS ELARM.....	60
FIGURE 66 - CHP2 OBSERVER TABLE VIEWER ELARM.....	61
FIGURE 67 - CHP02 REBAR3 AS PROD TAR.....	62
FIGURE 68 - CHP02 COPY CHP02-0.1.0.TAR.GZ TO MYREL.....	62
FIGURE 69 - CHP02 PROD CONSOLE FAILED TO RUN OBSERVER.....	63
FIGURE 70 - CHP02 INSTALL PROD RELEASE.....	64
FIGURE 71 - BIN/CHP02.CMD CONSOLE.....	64
FIGURE 72 - CHP02 PROD RELEASE CONSOLE OBSERVER.....	65
FIGURE 73 - OBSERVER PROCESSES TABLE.....	66
FIGURE 74 - CHP02 CONSOLE APPLICATION:WHICH_APPLICATIONS().	67
FIGURE 75 - CHP02 CONSOLE RAISE:ALARM().	68
FIGURE 76 - CHP02 TABLE VIEWER ETS:ALARMLIST TABLE.....	68
FIGURE 77 - CHP02 ELARM:RAISE(TEST, "SRC", [2]).	69
FIGURE 78 - CHP02 TABLE VIEWER ALARMLIST EDIT OBJECT DIALOG.....	70
FIGURE 79 - CHP02 CLEAR ALARM.....	70
FIGURE 80 - CHP02 TABLE VIEWER ALARMS CLEARED.....	71
FIGURE 81 - HISTOGRAM BAR CHART.....	72
FIGURE 82 - CHP02 REBAR3 PKGS FOLSOM.....	73
FIGURE 83 - CHP02 REBAR3 RELEASE MISSING FROM KERNEL.....	75
FIGURE 84 - CHP02_APP.ERL START FOLSOM APPLICATION.....	76
FIGURE 85 - CHP02 FOLSOM METRICS.....	77
FIGURE 86 - CHP02 REBAR3 PKGS RECON.....	77
FIGURE 87 - CHO02 REBAR3 RELEASE.....	78
FIGURE 88 - TRACING VENN DIAGRAM.....	79
FIGURE 89 - RECON_TRACE CALLS.....	80
FIGURE 90 - CHP02 REBAR3 TREE.....	80
FIGURE 91 - LISTING OF THE ~/.CONFIG/REBAR3/TEMPLATES DIRECTORY.....	83
FIGURE 92 - REBAR3 NEW BASIS_CLUSTER CHP03 COMMAND.....	84
FIGURE 93 - MAKE DEVREL COMMAND.....	85
FIGURE 94 – MAKE DEV1-CONSOLE, MAKE DEV2-CONSOLE, MAKE DEV3-CONSOLE.....	85
FIGURE 95 - NET_KERNEL:CONNECT_NODE CALLS TO CHP03_DEV2 AND CHP03_DEV3.....	86
FIGURE 96 - [NODE() NODES()] LIST APPEND.....	86
FIGURE 97 - CAT COMMAND ON ALL THREE DEV RELEASES.....	87
FIGURE 98 - RPC CALL FROM CHP03_DEV2 TO CHP03_DEV3.....	88
FIGURE 99 - REBAR3 NEW BASIS_CLUSTER_FAILOVER CHP03 COMMAND.....	88
FIGURE 100 - MAKE DEVREL COMMAND.....	89
FIGURE 101 – CHP03 “MAKE DEV1-CONSOLES;MAKE DEV2-CONSOLE;MAKE DEV3-CONSOLE” COMMANDS.....	89
FIGURE 102 - OBSERVER CHP03_DEV1, CHP03_DEV2, AND CHP03_DEV3.....	90
FIGURE 103 - APPLICATION:TAKEOVER(CHP03, PERMANENT).....	91
FIGURE 104 - CHP03_DEV1 AFTER TAKEOVER FROM CHP03_DEV2.....	91
FIGURE 105 - CHP03_DEV2 ACTIVE AFTER TAKEOVER.....	91
FIGURE 106 - CHP03_DEV2 SIMULATE FAILOVER.....	92
FIGURE 107 - CHP03_DEV1 ACTIVE AFTER FAILOVER ON CHP03_DEV2.....	92

LIST OF TABLES

TABLE 1 HYPOTHETICAL PRODUCT CAPABILITIES 8

TABLE 2 – LOG LEVELS.....52

TABLE 3 - NODES() RESULTS ON ALL THREE NODES 86

LIST OF LISTINGS

LISTING 1 – REBAR3.CONFIG, CONFIG/SYS.CONFIG, AND CONFIG/VM.ARGS FILES	35
LISTING 2 – CHP01.APP.SRC	36
LISTING 3 - CHP01_APP.ERL.....	36
LISTING 4 - CHP01_SUP.ERL	37
LISTING 5 - UPDATED CHP01_SUP.ERL	38
LISTING 6 – BIN/CHP01.CMD LIST COMMAND.....	44
LISTING 7 - SASL CONFIG	48
LISTING 8 - REBAR.CONFIG ADD OVERLAY COMMAND	49
LISTING 9 - LAGER PACKAGE DEPENDENCY	49
LISTING 10 - LAGER CONFIG FILES.....	50
LISTING 11 – SAMPLE CRASH FILE	51
LISTING 12 – LAGER LOGGING FUNCTION	53
LISTING 13 - ADD ELARM REPO TO REBAR.CONFIG	58
LISTING 14 - CHP02 PROD REBAR.CONFIG UPDATED FOR OBSERVER APPLICATION	63
LISTING 15 - CHP02 REBAR.CONFIG ADD FOLSOM REPO.....	73
LISTING 16 - CHP02 SYS.CONFIG ADD THE FOLSOM APPLICATION CONFIGURATION	74
LISTING 17 - CHP02 REBAR.CONFIG ADD BEAR DEPENDENCY	75
LISTING 18 – CHP02 REBAR3.CONFIG ADD RECON REPO.....	78
LISTING 19 - CHP03_DEV1 SYS.CONFIG FILE	93

PREFACE

As a software developer I'm a big believer in the use of frameworks to increase productivity, quality and provide a commonality between developers working together on a product. I've also become increasingly aware in recent years of the need to provide solutions that allow for the monitoring and tracing of an online production system that is distributed from the start. *"Our customers want a Toaster, which just runs until the hardware fails -- to coin a phrase"*. The proceeding phrase correctly describes our customers' expectation for the turn-key solutions we provide.

The Erlang Bootcamp is designed to accelerate the learning curve for adopting Erlang as a new framework, language, environment, and virtual machine as well as a brief introduction to the functional programming world for programmers who currently only have experience with imperative languages, such as C/C++, C# or Java¹.

2. INTRODUCTION

2.1 WHY ERLANG

In the field of software engineering there are so many frameworks to choose from, it gets difficult to decide on the technology in which to base your future on. The overall popularity of a framework and language should not be the deciding factor on whether or not it should be used. Instead the criteria should be based on what the capabilities the framework provides and the qualities it brings to your solution.

Let's take a hypothetical situation, where your management has asked you to create a new product for the company, one that doesn't exist but requires the following capabilities listed in Table 1:

Table 1 Hypothetical Product Capabilities

Characteristics	Description
Concurrency	Capability to support several thousand events simultaneously
Maximize Processes	Ability to run hundreds of thousands of processes
Process Isolation	Isolation via message passing, independent process memory management
Message Passing	Location transparency, local and remote node message passing
Fault Tolerance	Supervision of groups of processes and automatic recovery
Hot Code Loading	Allows upgrading/downgrading of processes without downtime
Distribution/Clusters	Applications distributed across a set of servers in a cluster
Database	Atomicity, Consistency, Isolation, Durability (ACID)-compliant distributed in memory and/or disk database
Open Source	Leverages intellectual work from open source communities and decreases time to market.
Platform Neutral	Runs on Linux, Windows, Macs, embedded devices such as Raspberry PI, AWS, and Azure, etc.
Web Technologies	Web Sockets, HTTP/2, SSE, JSON, etc.
Queuing Technologies	AMQP, MQTT, WAMP, etc.
CRDT/Vector Clocks	Conflict-free replicated data types

¹ Note that recent releases of C++, C# and Java now have functional programming capabilities with the inclusion of lambda expressions and tail call optimization.

The hypothetical requirements listed above actually represent what a most modern turnkey solutions need to support to fulfill customer's needs, aka the software equivalent of a Toaster. Joe Armstrong describes it in his Programming Erlang, 2nd Edition book as "*Once started, Erlang/OTP applications are expected to run forever*" [1]

While there are many frameworks available today, very few handle most of the hypothetical product requirements listed and the ones that do, have dependencies on an array of services and libraries that have to be carefully managed. Erlang is unique as it is a framework, a language, an environment, and a virtual machine that fulfills our hypothetical product capabilities.

The Erlang Bootcamp skips basic Erlang programming skills and jumps straight into the production of a series of Erlang OTP applications that progressively demonstrate the capabilities while educating the developer on the steps to creating a basis for any hypothetical product.

2.2 ERLANG DESIGN GOALS

The following is a quote from Joe Armstrong's white paper "A History of Erlang" describes at a high level the designed goals for the Erlang environment.

Erlang was designed for writing concurrent programs that "run forever." Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Erlang processes are lightweight and belong to the language, not the operating system. Erlang has mechanisms to allow programs to change code "on the fly" so that programs can evolve and change as they run. These mechanisms simplify the construction of software for implementing non-stop systems. [2]

2.3 ERLANG PROGRAMMING LANGUAGE

The following is an explanation from Ulf Wiger's "Four-fold Increase in Productivity and Quality" on page 7 on the origins of the Erlang as a programming language as it draws features from both concurrent and functional programming languages.

Erlang can be described as a concurrent functional language, combining two main traditions:

- *Concurrent programming languages: Modula, Chill, Ada, etc., from which Erlang inherits modules, processes, and process communication.*
- *Functional and logic programming languages: Haskell, ML, Miranda, Lisp, etc. from which Erlang inherits atoms, lists, guards, pattern matching, catch and throw etc.*

Significant design decisions behind the development of Erlang were:

- *It should be built on a virtual machine which handles concurrency, memory management etc., thus making the language independent of the operating system, and thus more portable.*
- *It should be a symbolic language with garbage collection, dynamic typing, and with data types like atoms, lists and tuples.*
- *It should support tail recursion, so that even infinite loops can be handled by recursion.*
- *It should support asynchronous message passing and a selective receive statement.*
- *It should enable default handling of errors, enabling an aggressive style of programming. [3]*

3. INSTALLATIONS

The following sections cover the installation of Erlang OTP 22.1, Git 2.24.1.2 and Rebar3 installations that will be used for the Erlang Bootcamp exercises.

3.1 INSTALLATION OF ERLANG OTP 22.1

The following section describes the steps to install the Erlang on a Windows 64 bit machine.

1. To download the Erlang OTP 22.1 go to the following link:
https://erlang.org/download/otp_win64_22.1.exe
2. Run the otp_win64_22.1.exe executable.
 - a. Click the Run button in the **Open File – Security Warning** dialog as shown in Figure 1.

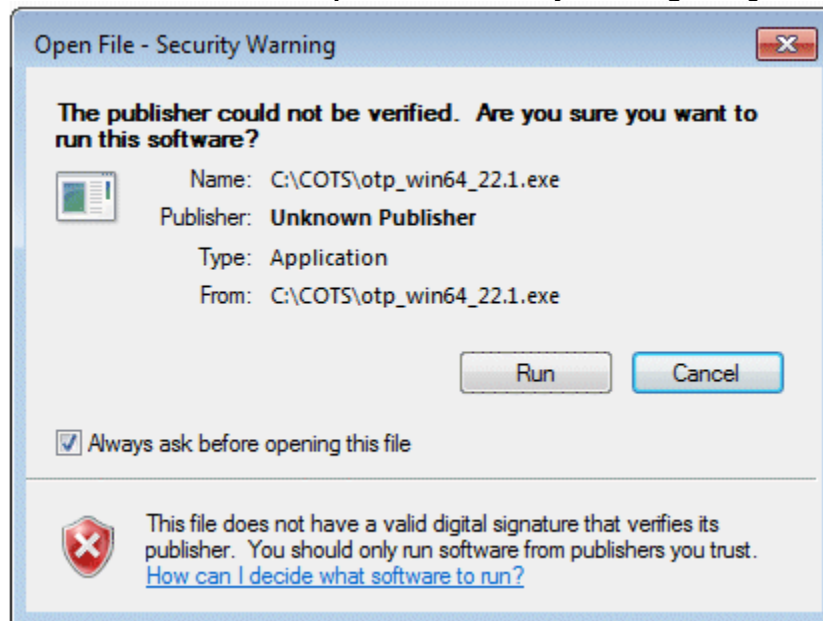


Figure 1 - Open File - Security Warning

- - b. Click the checkbox in for Microsoft DLLs in the **Choose Components** dialog as shown in Figure 2.

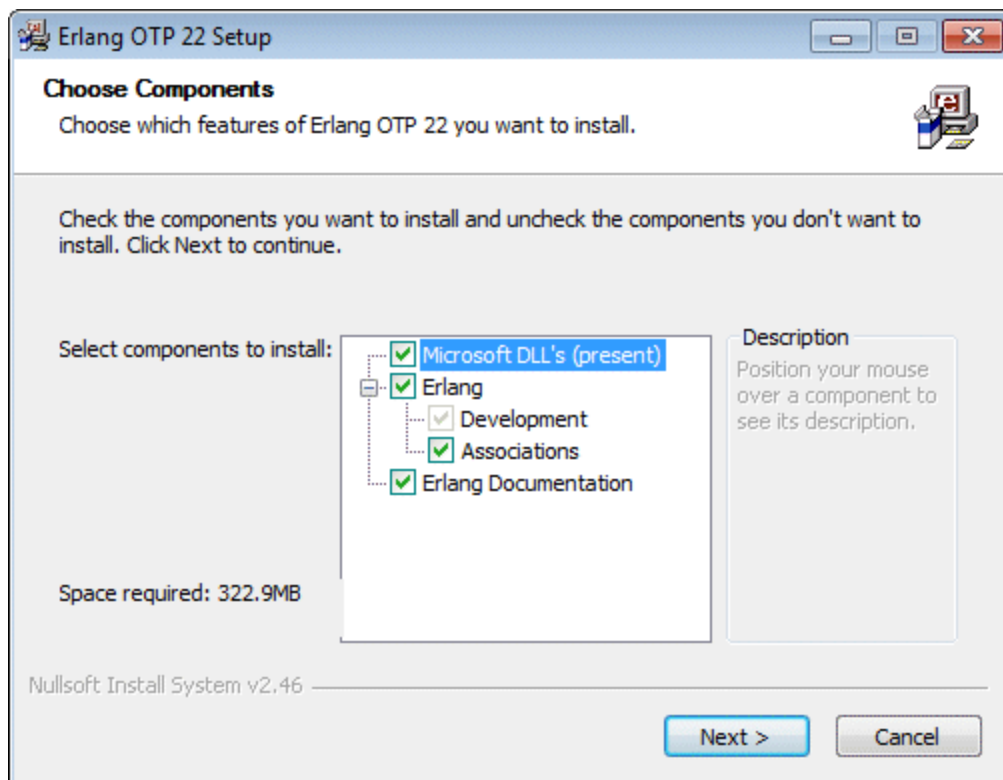


Figure 2 - Choose Components

- c. Optionally change the designated drive and click the Next button in the **Choose Install Location** dialog as shown in Figure 3.

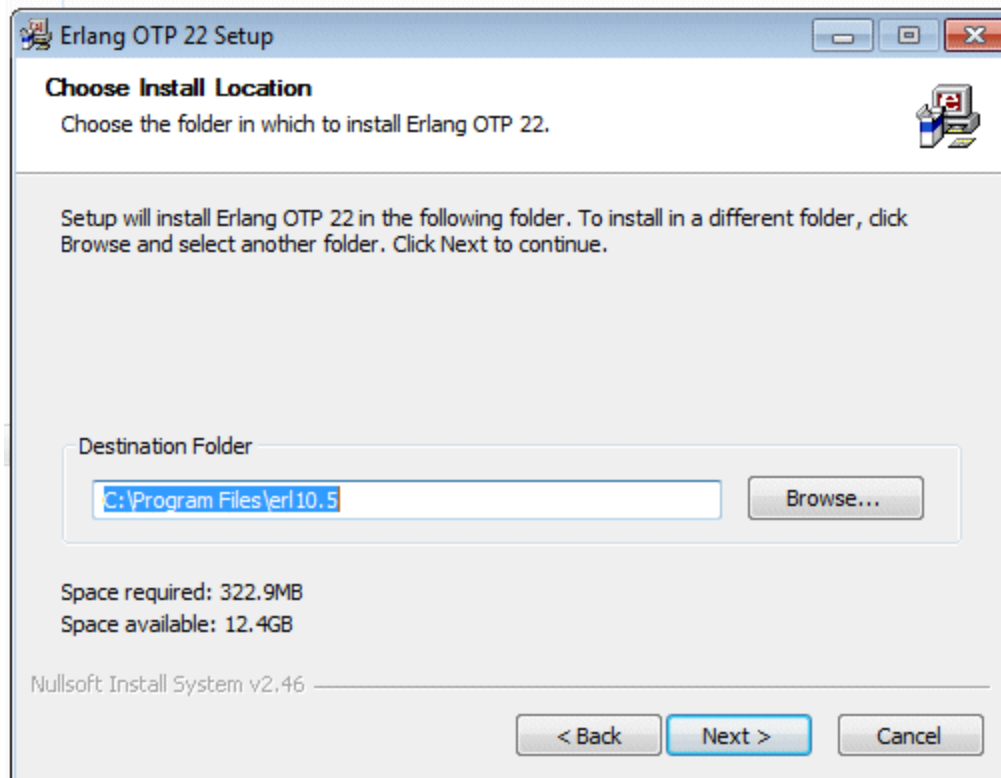


Figure 3 - Choose Install Location

- d. Optionally change the Startup Menu Folder and click the Install button in the **Choose Startup Menu Folder** dialog as shown in the Figure 4.

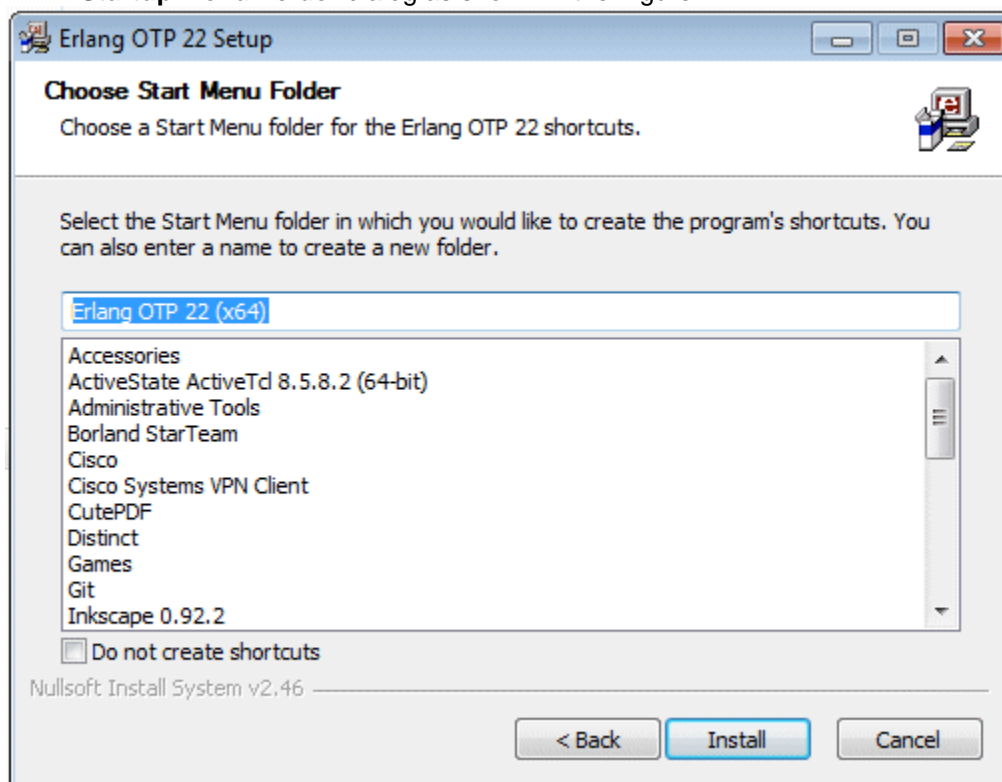
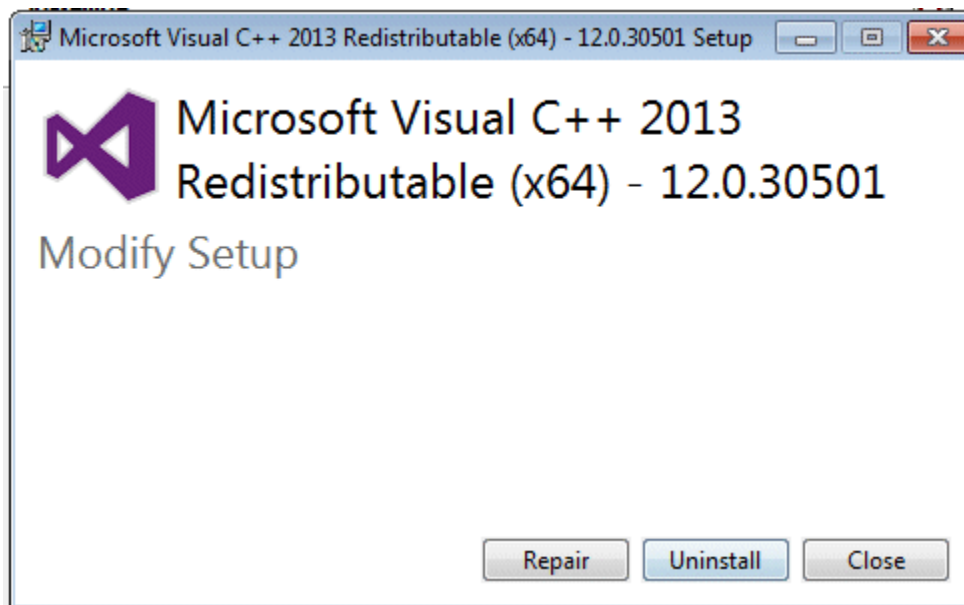


Figure 4 - Choose Start Menu Folder

- e. Dependent on your machine install, you may be prompted to install or repair the Microsoft Visual C++ 2013 Redistributables as show in Figure 5.

**Figure 5 - Install/Repair Microsoft Visual C++ Redistributables (x64)**

- f. Click the Close button in the **Installation Complete** dialog as show in Figure 6.

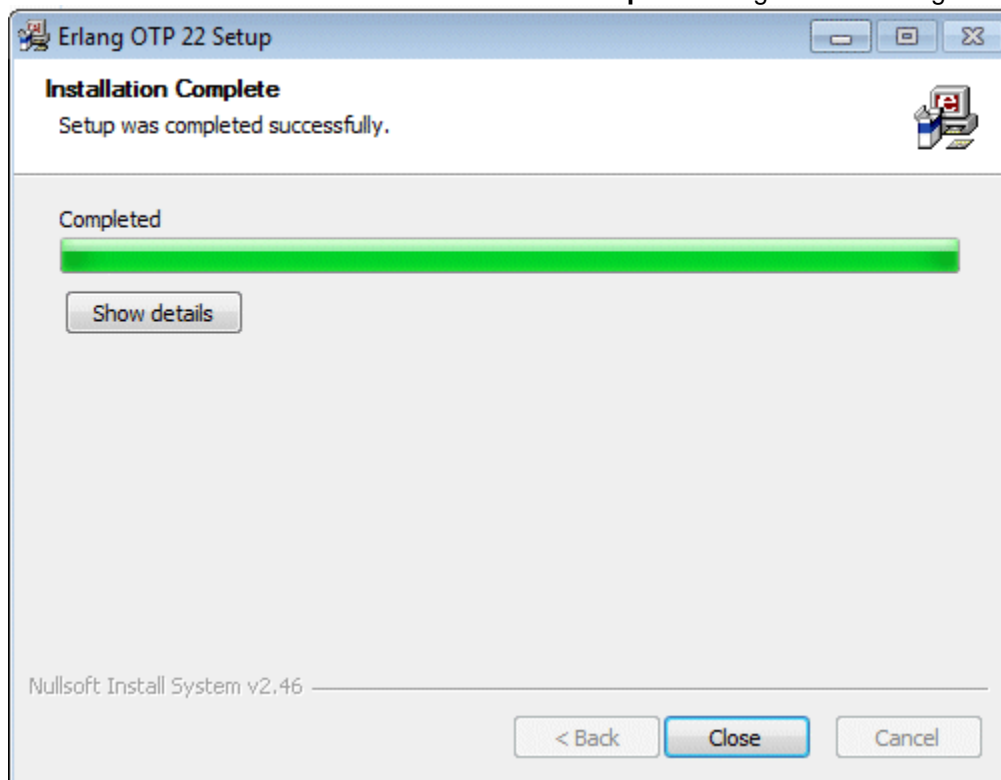
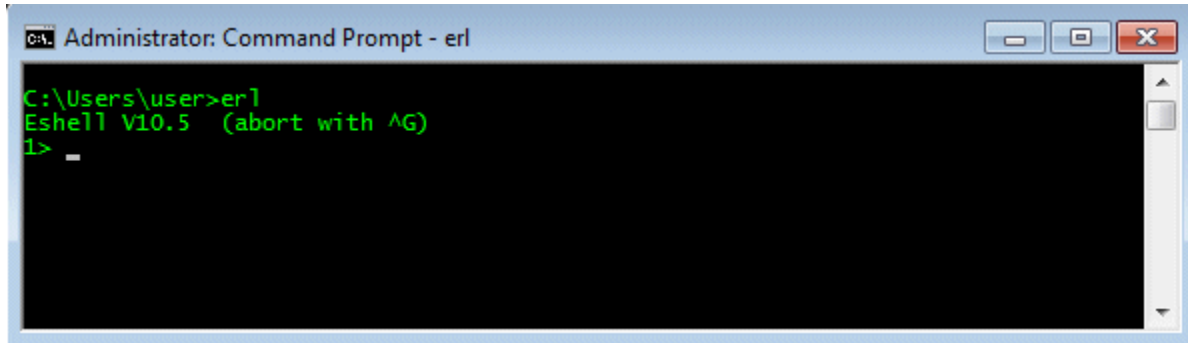


Figure 6 – Installation Complete

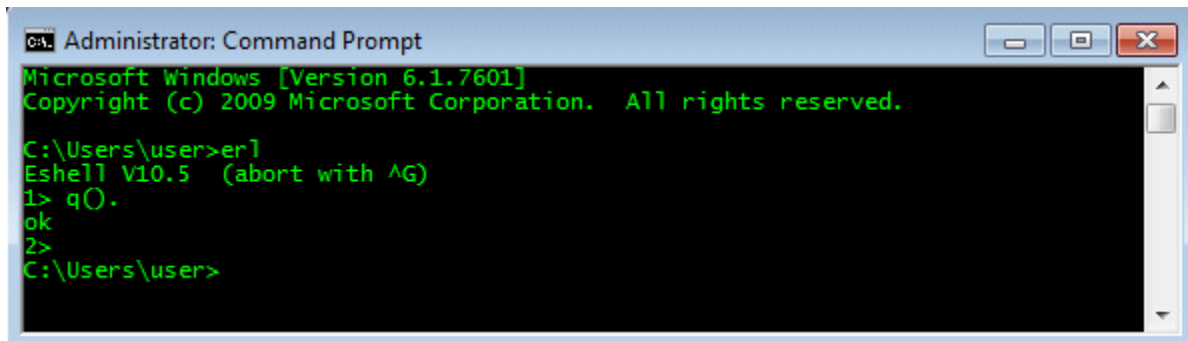
3. After the software installation of Erlang OTP 22.1, add to the system path the Erlang bin folder, e.g., "C:\Program Files\erl10.5\bin;"
4. Step #3 above makes it possible to run the Erlang shell from a Windows **cmd** prompt. To verify the path is correct, start a new cmd prompt window and type `erl` and return to start the Eshell session as show in Figure 7.



```
C:\Users\user>erl
Eshell V10.5 (abort with ^G)
1>
```

Figure 7 - Eshell V10.5 session

5. Next, we need to exit out of the Eshell session, type "`q()`." at the `1>` prompt without the quotes, this is known as a quit function call, which is used to exit out of a Eshell session. Figure 8 shows the results of the `q()` function call, which returns to the Windows **cmd** prompt session.



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\user>erl
Eshell V10.5 (abort with ^G)
1>q()
ok
2>
C:\Users\user>
```

Figure 8 - Eshell Quit session

3.2 GIT

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. We will be using Git for cloning open source repositories and management of own repositories.

3.2.1 Installation of Git 2.24.1.2

The following section describes the steps to install the Git 2.24.1.2 on a Windows 64 bit machine.

1. To download the Git-2.24.1.2-64-bit go to the following link:

<https://git-scm.com/download/win>

2. Run the Git-2.24.1.2-64-bit.exe executable.
 - a. Click the Next button in the **Git 2.24.1.2 Setup Information** dialog as shown in Figure 9.



Figure 9 - Git 2.24.1.2 Setup Information

- b. Optionally change the destination location and click the Next button in the **Git 2.24.1.2 Setup Select Destination Location** dialog as shown in Figure 10.

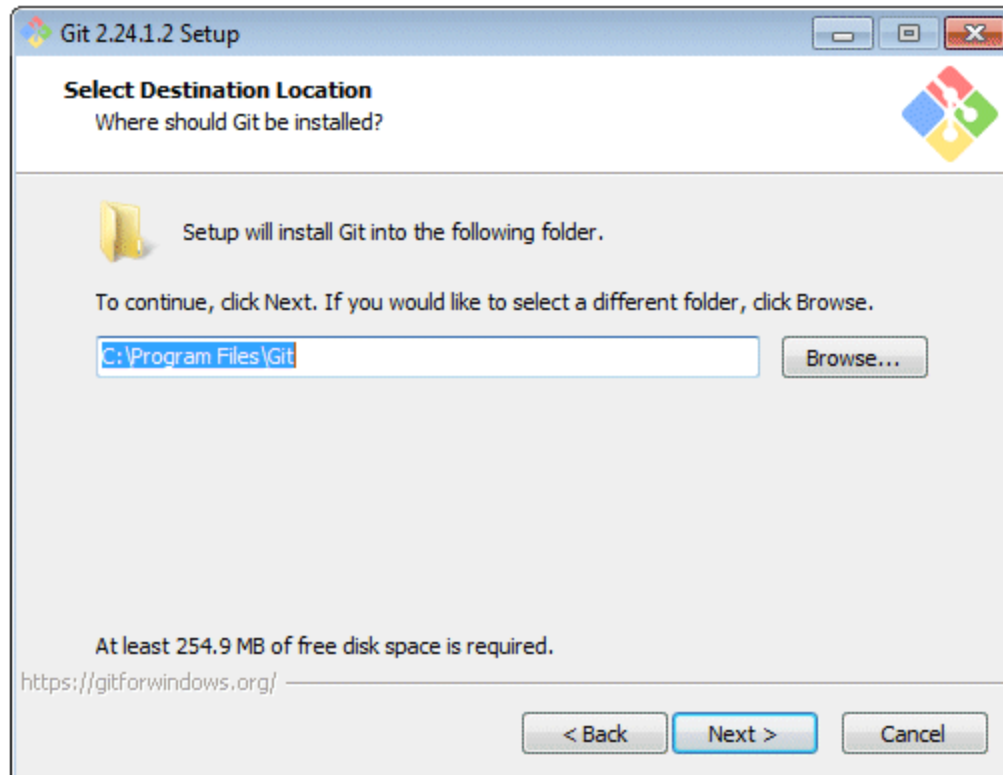


Figure 10 - Git 2.24.1.2 Setup Select Destination Location

- c. Optionally select components to install and click the Next button in the **Git 2.24.1.2 Setup Select Components** dialog as show in Figure 11.

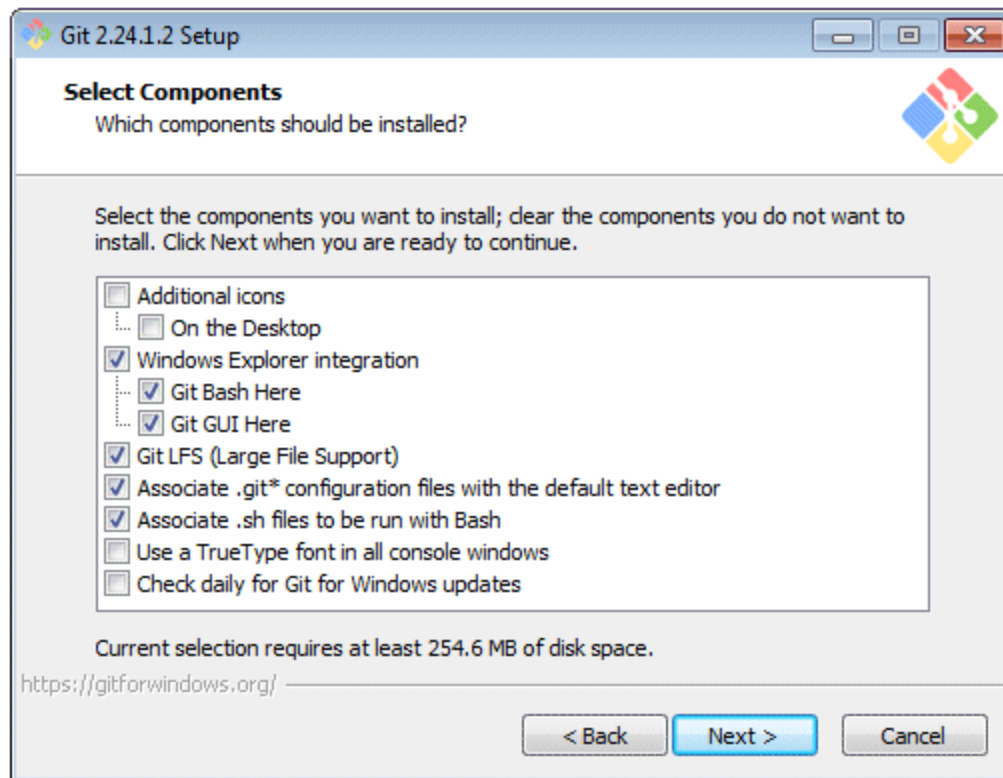


Figure 11 - Git 2.24.1.2 Setup Select Components

- d. Click the Next button in the **Git 2.24.1.2 Setup Select Start Menu Folder** dialog as show in Figure 12.

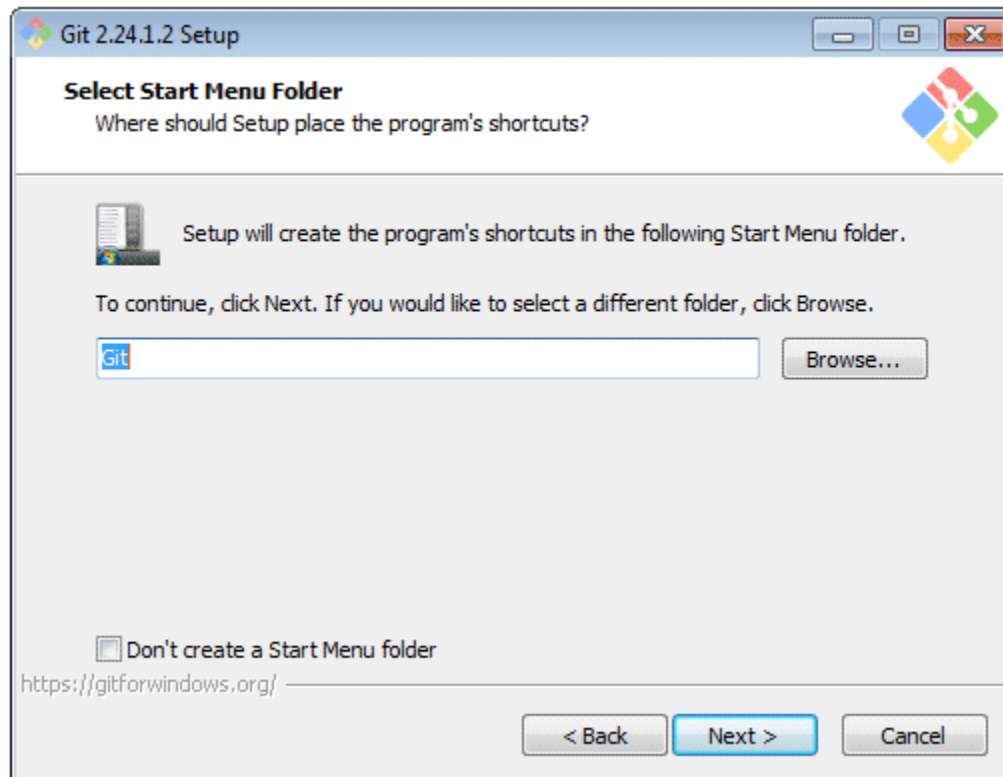


Figure 12 - Git 2.24.1.2 Setup Select Start Menu Folder

- e. Choose the default editor and click the Next button in the **Git 2.24.1.2 Setup Choosing the default editor used by Git** dialog as shown in Figure 13.

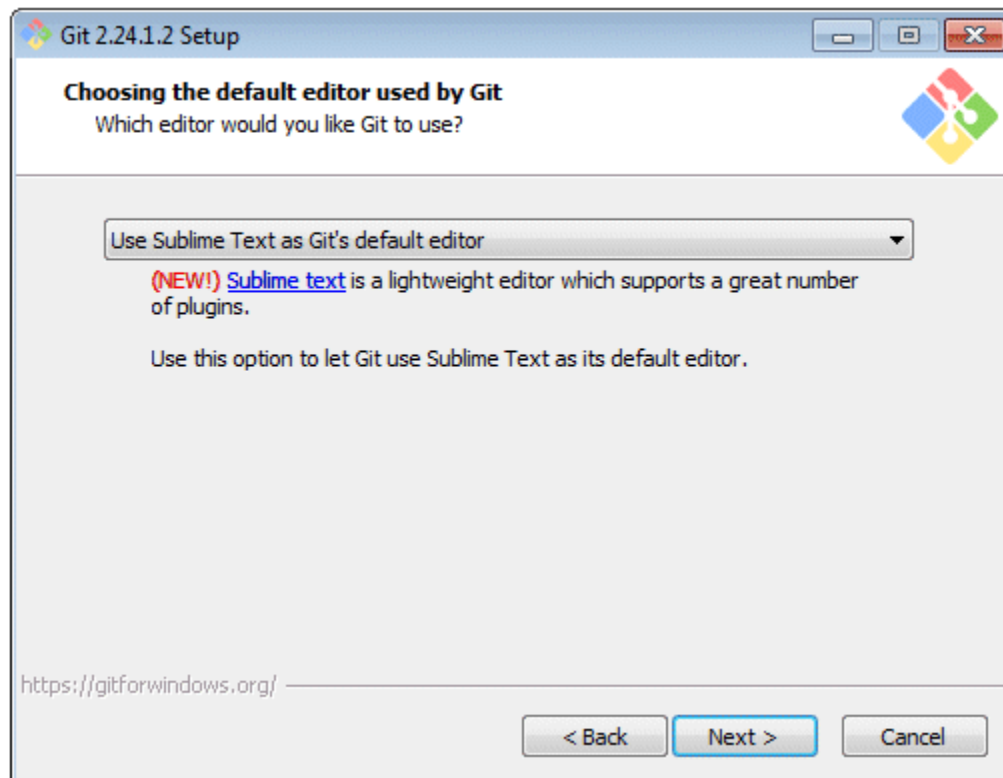


Figure 13 - Git 2.24.1.2 Setup Choosing the default editor used by Git

- f. Click the Next button in the **Git 2.24.1.2 Setup Adjusting your PATH environment** dialog as shown in Figure 14.

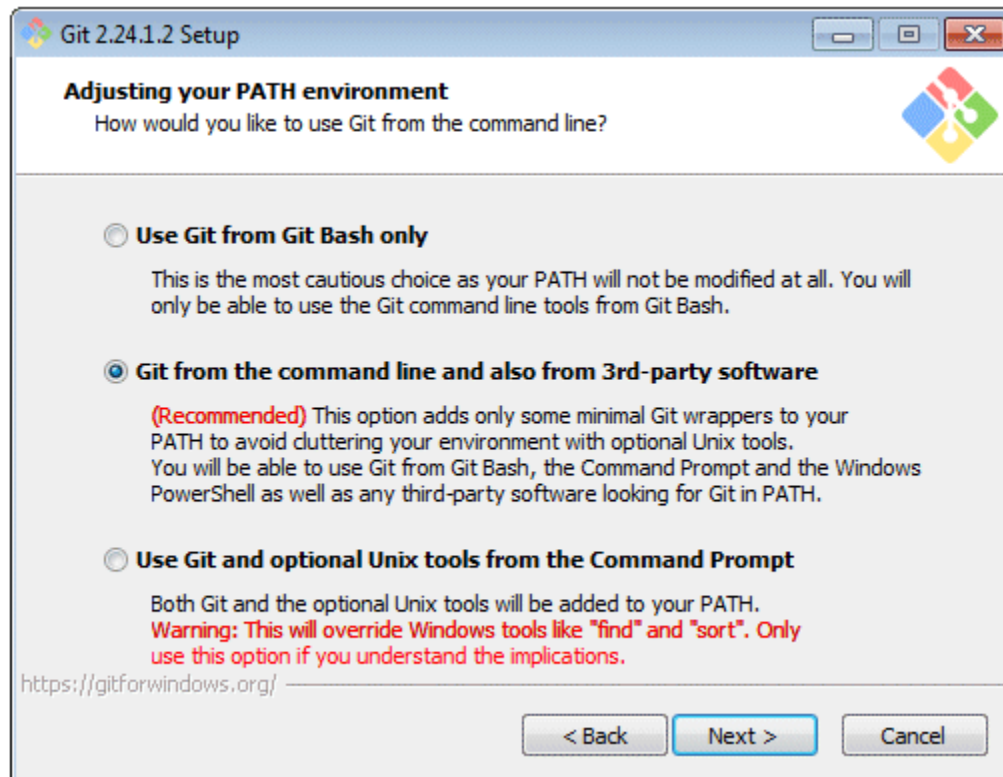


Figure 14 - Git 2.24.1.2 Setup Adjusting your PATH environment

- g. Click the Next button in the **Git 2.24.1.2 Setup Choosing HTTPS transport backend** dialog as show in Figure 15.

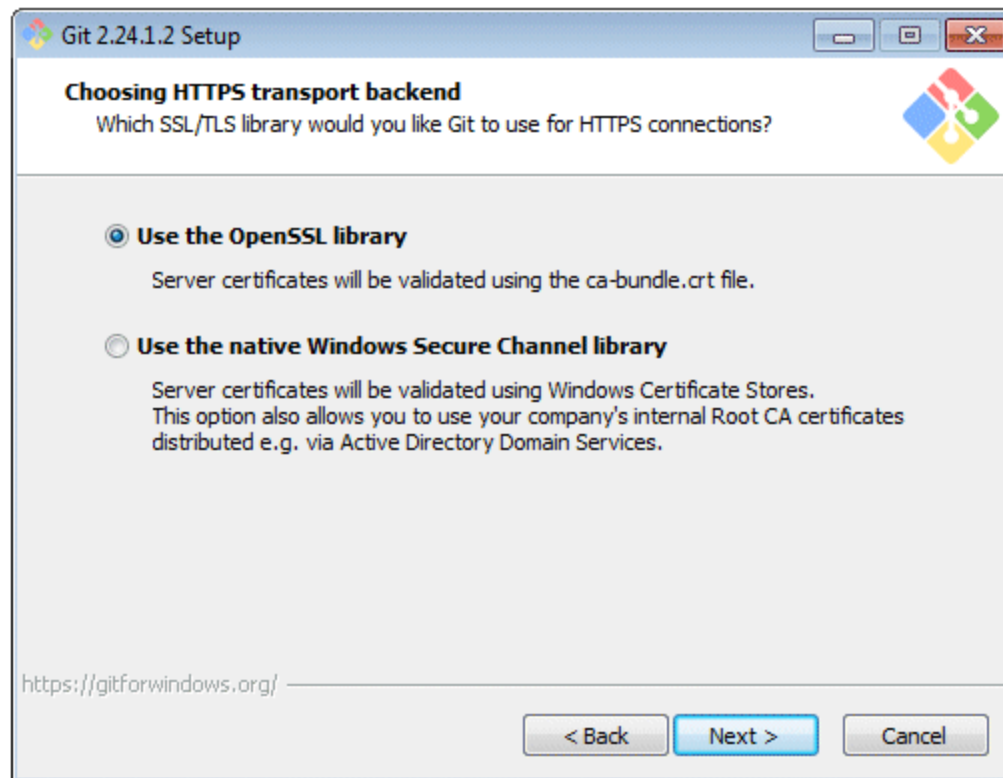


Figure 15 - Git 2.24.1.2 Setup Choosing HTTPS transport backend

- h. Click the Next button in the **Git 2.24.1.2 Setup Configuring the line ending conversions** dialog as show in Figure 16.

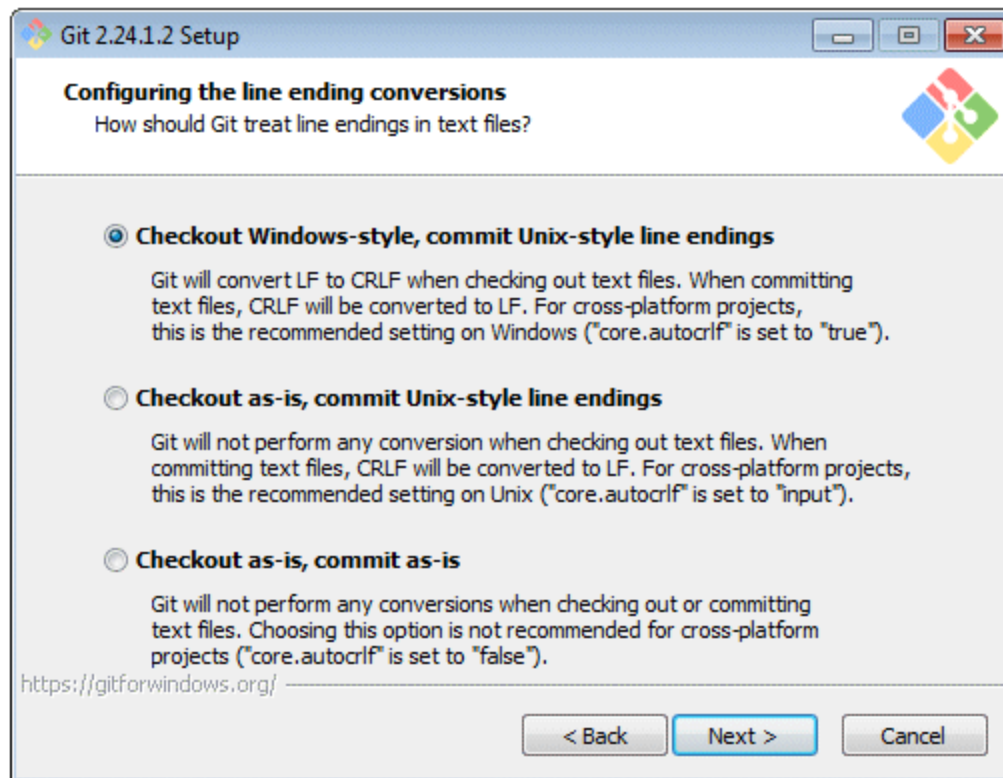


Figure 16 - Git 2.24.1.2 Setup Configuring the line ending conversions

- i. Click the Next button in the **Git 2.24.1.2 Setup Configuring the terminal emulator to use with Git Bash** as show below in Figure 17.

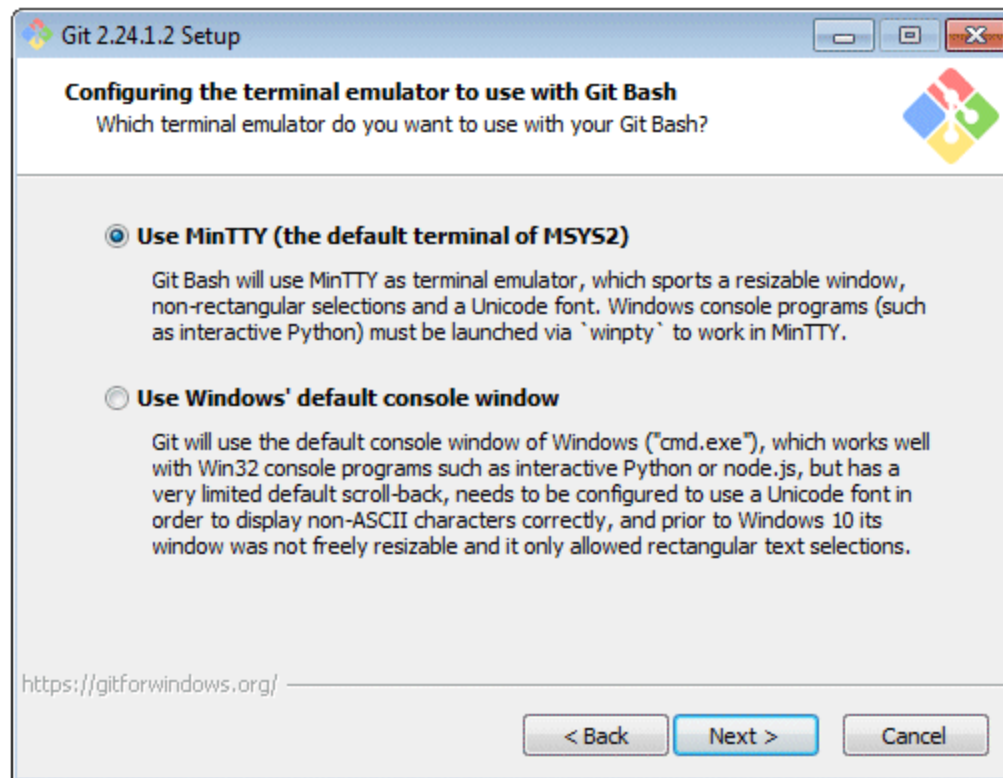


Figure 17 - Git 2.24.1.2 Setup Configuring the terminal emulator to use with Git Bash

- j. Click the Next button in the **Git 2.24.1.2 Setup Configuring extra options** dialog as show in Figure 18.

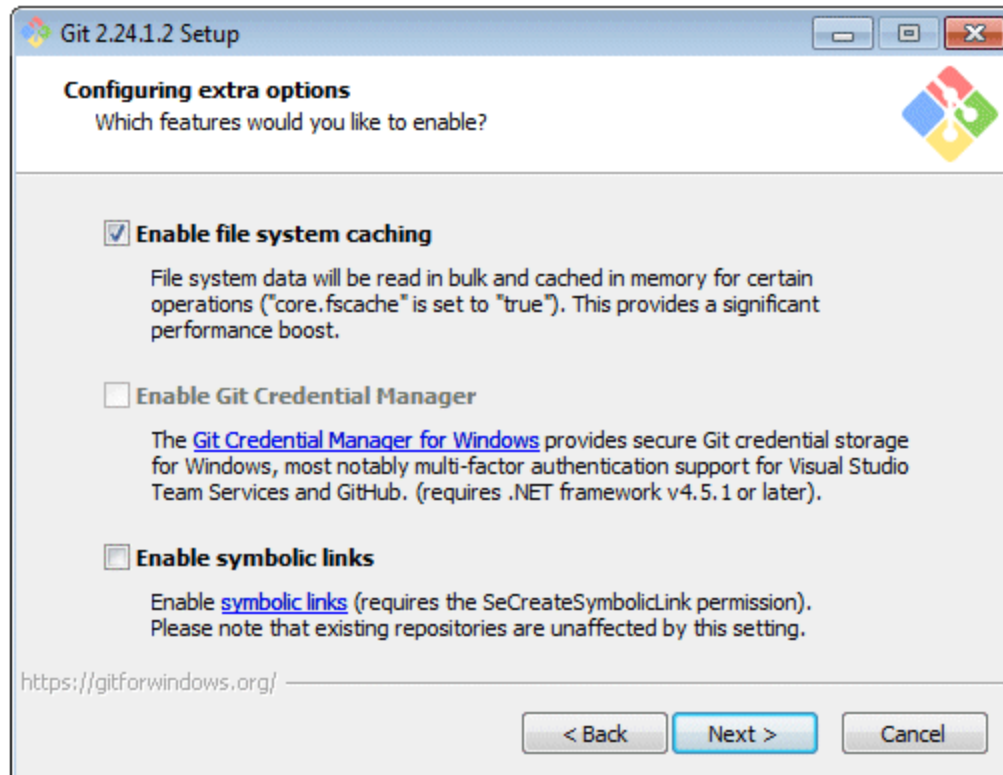


Figure 18 - Git 2.24.1.2 Setup Configuring extra options

- k. Click the Install button in the **Git 2.24.1.2 Setup Configuring experimental options** as show in Figure 19.

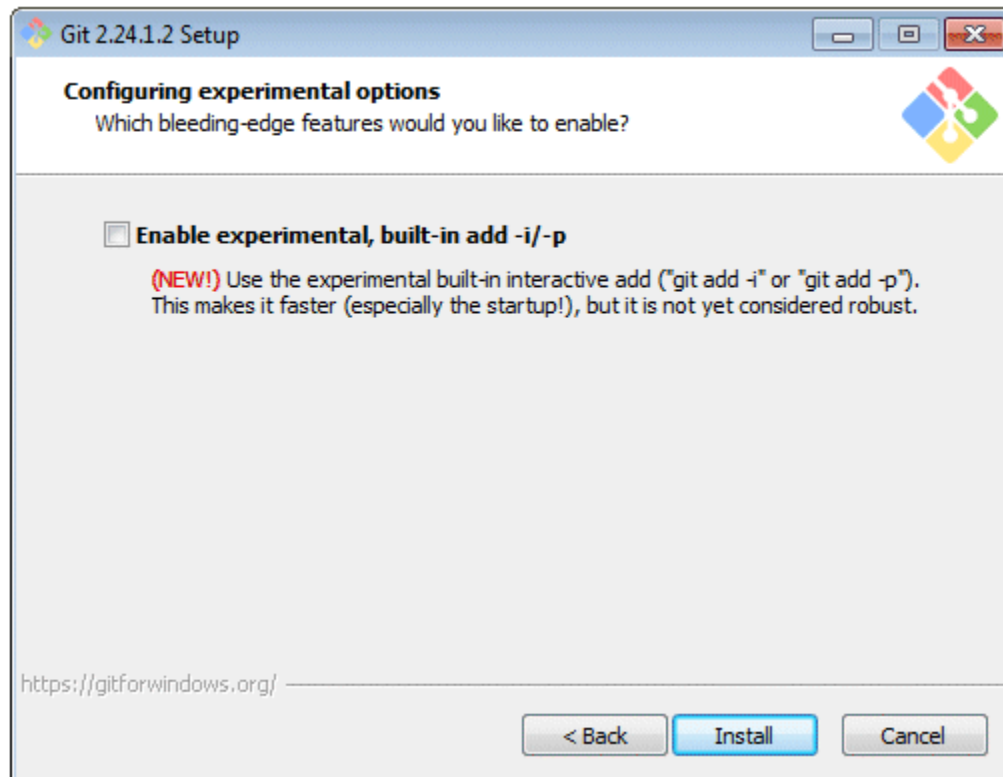


Figure 19 - Git 2.24.1.2 Setup Configuring experimental options

- I. Click the Finish button in the **Git 2.24.1.2 Setup Completing the Git Setup Wizard** dialog as shown in Figure 20.

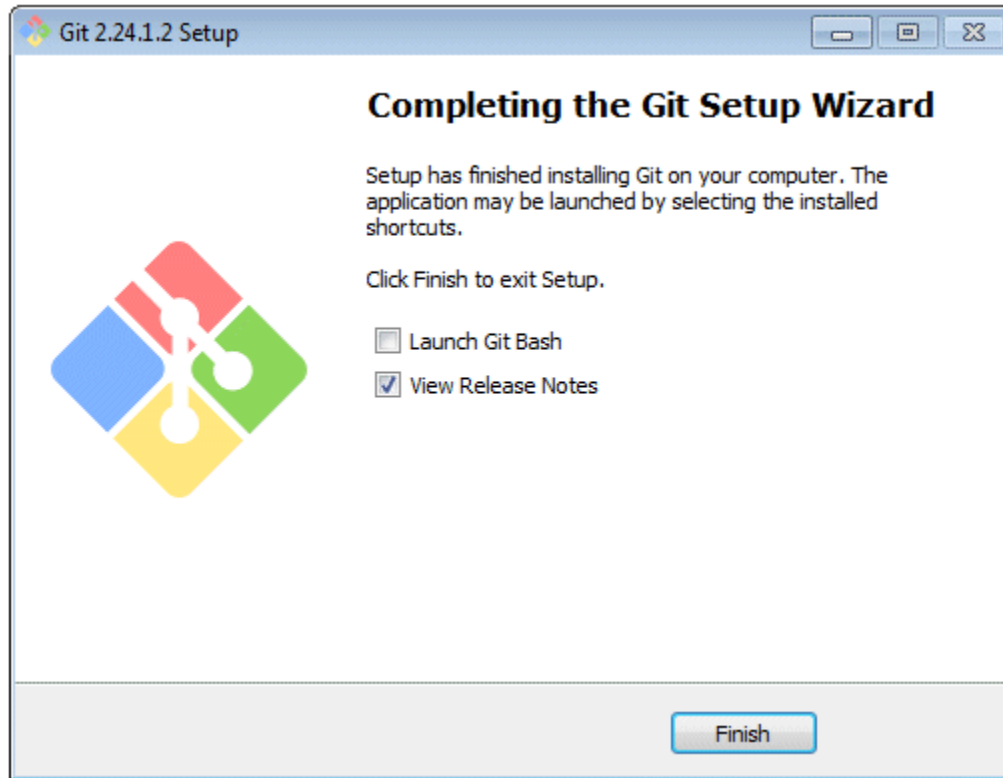


Figure 20 - Git 2.24.1.2 Setup Completing the Git Setup Wizard

3.3 GNU MAKE FOR WINDOWS

The GNU Make for Windows adds the capability for adding Makefile scripts as part of your developmental scripts for building Erlang OTP applications. The Erlang Bootcamp makes use of the Make capability typically calling sets of Rebar3 calls and other miscellaneous development actions as part of the overall build environment.

3.3.1 Installation of GNU Make for Windows

The following section describes the steps to install the GNU Make for Windows.

1. Download form the binaries from the following link <http://gnuwin32.sourceforge.net/downloads/make-bin-zip.php> and the dependencies from <http://gnuwin32.sourceforge.net/downloads/make-dep-zip.php> into a temp directory and extract the contents of the zip archives. Copy the make.exe, libintl3.dll, and libiconv2.dll to the "C:\Program Files\Git\Mingw64\bin" folder.

3.4 REBAR3

The Rebar3 application is defacto standard way to build Erlang OTP applications and is for starting new projects as well management of Configuration, Dependencies, Profiles, Running of Tests, Releases and compilation of Erlang source code. We will be learning more about Rebar3 by example in the Bootcamp chapters; the following is a link to the Rebar3 documentation that you can use as a reference <https://www.rebar3.org/docs>.

3.4.1 Installation of Rebar3

The following section describes the steps to install Rebar3 on a Windows 64 bit machine.

1. Use Git from the Git Bash shell to clone the <https://github.com/jlstarnes/rebar3.git> into C:\erlang\tools\rebar3 folder as show in Figure 21 below. Note we are using my repository for Rebar3 due to an issue in relx v3.30.0 that I found during the writing of the Erlang Bootcamp. This repository is forked off the Rebar3 master currently at version 3.13.0 and temporarily replaces the default relx v3.30.0. I have submitted a problem report to the maintainers of relx, which may take some time to show up in the relx master.

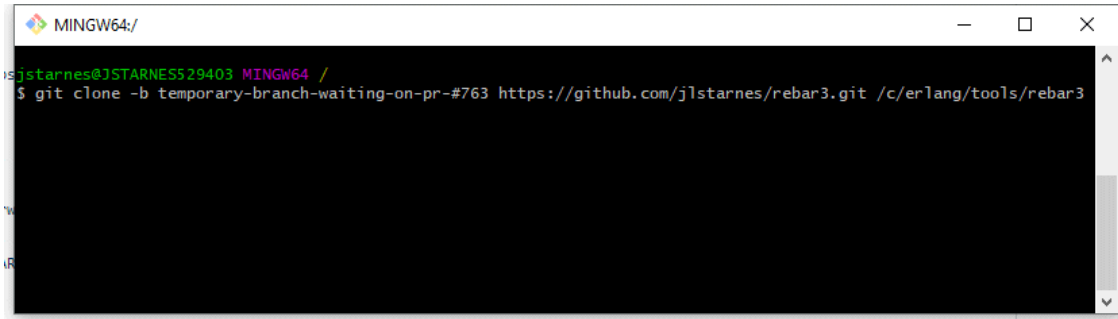


Figure 21 - Git Clone of Rebar3 repository

2. Figure 22 shows the result of the “git clone –b temporary-branch-waiting-on-pr-#763 <https://github.com/jlstarnes/rebar3.git> /c/erlang/tools/rebar3” command.

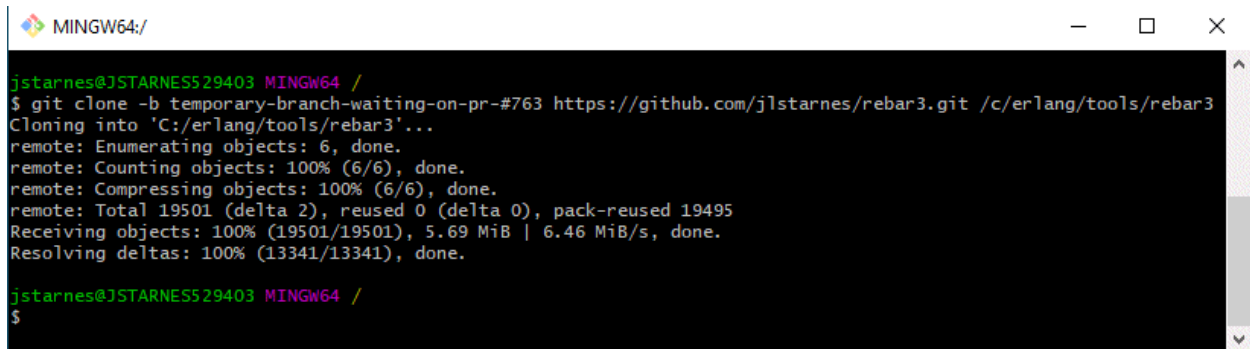


Figure 22 - Git Cloned of Rebar3 Completed

3. In the Git Bash shell change directory to “/c/Erlang/tools/rebar3” as shown below in Figure 23.

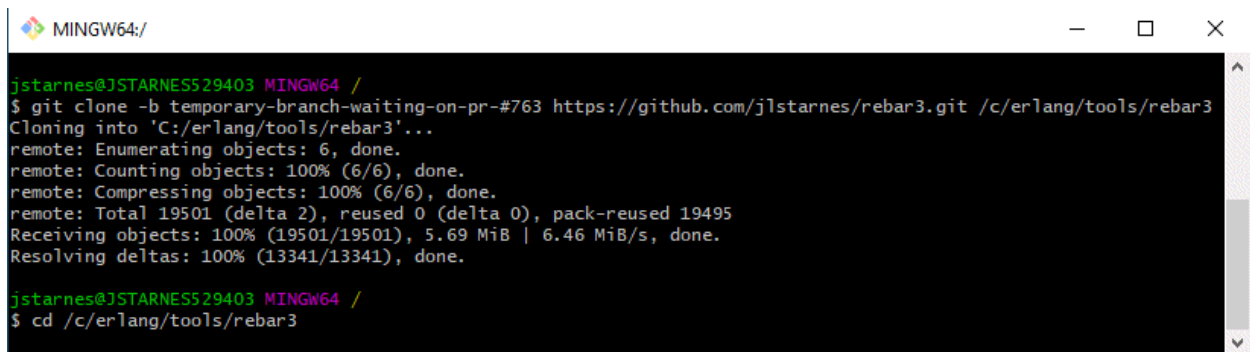
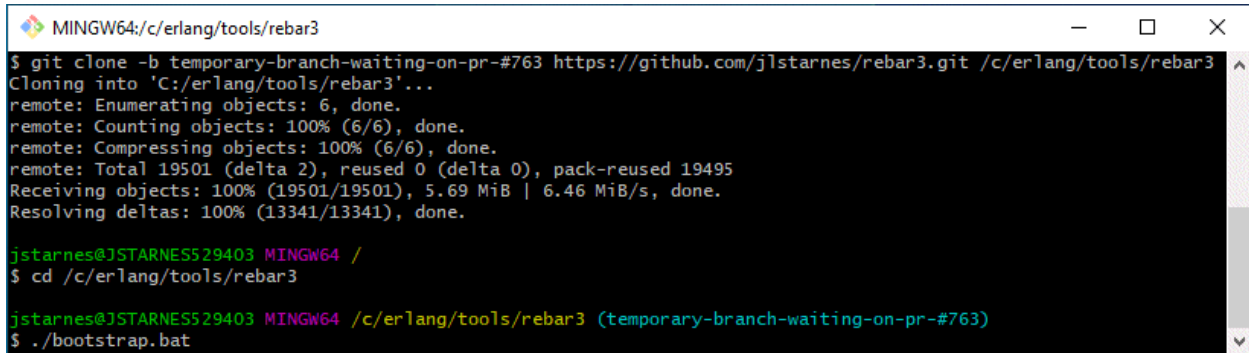


Figure 23 - Change Directory to Rebar3 folder

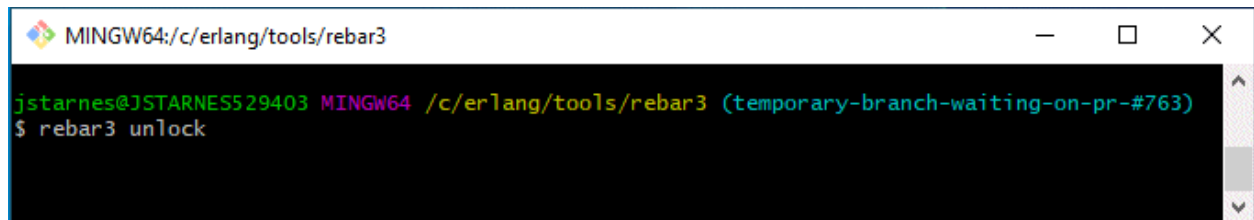
4. In the Git Bash shell type “./bootstrap.bat” on the command line prompt without the quotes and press the return key. This command builds the rebar3 and rebar3.cmd by pulling in additional remote repositories as shown in Figure 24.



```
MINGW64:/c/erlang/tools/rebar3
$ git clone -b temporary-branch-waiting-on-pr-#763 https://github.com/jlstarnes/rebar3.git /c/erlang/tools/rebar3
Cloning into 'C:/erlang/tools/rebar3'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 19501 (delta 2), reused 0 (delta 0), pack-reused 19495
Receiving objects: 100% (19501/19501), 5.69 MiB | 6.46 MiB/s, done.
Resolving deltas: 100% (13341/13341), done.
jstarnes@JSTARNES529403 MINGW64 /
$ cd /c/erlang/tools/rebar3
jstarnes@JSTARNES529403 MINGW64 /c/erlang/tools/rebar3 (temporary-branch-waiting-on-pr-#763)
$ ./bootstrap.bat
```

Figure 24 - Rebar3 Bootstrap command

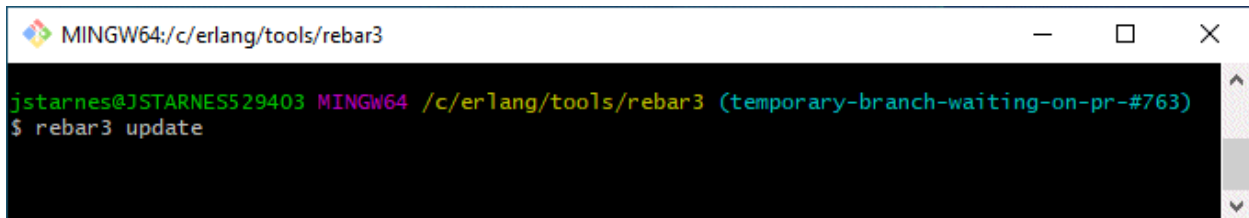
5. Since we are using our own private repository to temporarily get around a bug in the relx application that is a dependency of the rebar3 application, we have to do some extra steps in order for the bootstrap command to pull correct sub-dependencies. This is because the dependencies repositories of rebar3 are typically locked and we need to unlock them otherwise the default relx 3.30.0 repository is used and the changed repository of <https://github.com/jlstarnes/relx.git> branch “Fix-set_erts_dir_from_erl” will log in debug output and be ignored.
 - a. After the running the “./bootstrap.bat” command, we need to do the rebar3 unlock command as show in Figure 25.



```
MINGW64:/c/erlang/tools/rebar3
jstarnes@JSTARNES529403 MINGW64 /c/erlang/tools/rebar3 (temporary-branch-waiting-on-pr-#763)
$ rebar3 unlock
```

Figure 25 - Rebar3 unlock command

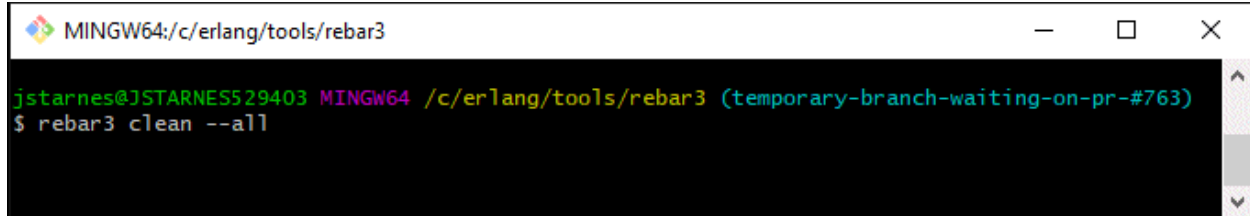
- b. After running the “rebar3 unlock” command we need to do the rebar3 update command as show in Figure 26.



```
MINGW64:/c/erlang/tools/rebar3
jstarnes@JSTARNES529403 MINGW64 /c/erlang/tools/rebar3 (temporary-branch-waiting-on-pr-#763)
$ rebar3 update
```

Figure 26 - Rebar3 update command

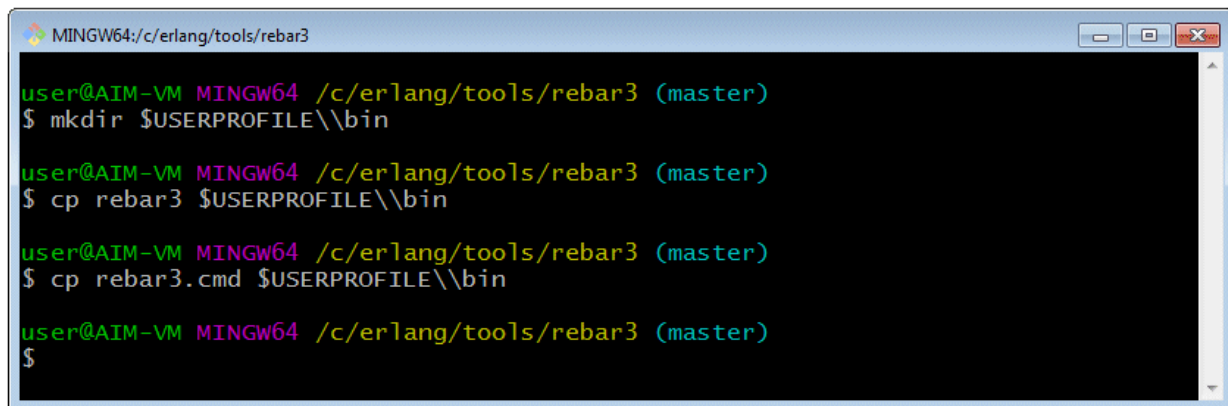
- c. After running the “rebar3 update” command we need to do the rebar3 clean -all command as show in Figure 27.



```
MINGW64:/c/erlang/tools/rebar3
jstarnes@JSTARNES529403 MINGW64 /c/erlang/tools/rebar3 (temporary-branch-waiting-on-pr-#763)
$ rebar3 clean --all
```

Figure 27 - Rebar3 clean --all command

- d. After running the “rebar3 clean --all: command we need to redo the bootstrap command first shown in Figure 24. So it will pull in the repository of <https://github.com/jlstarnes/relx.git> branch “Fix-set_erts_dir_from_eri”
6. Next, from the current Git Bash shell in Figure 24 do the following three (3) steps, after the following three (3) steps your shell should be similar to Figure 28 below.
 - a. Create a bin folder underneath your \$USERPROFILE directory by typing in the following command at the \$prompt, i.e., “mkdir \$USERPROFILE\\bin”.
 - b. Copy the rebar3 file to the \$USERPROFILE\\bin folder by typing the following command at the \$prompt, i.e., “cp rebar3 \$USERPROFILE\\bin”.
 - c. Copy the rebar3.cmd file to the \$USERPROFILE\\bin folder by typing the following command at the \$prompt, i.e., “cp rebar3.cmd \$USERPROFILE\\bin”.



```
MINGW64:/c/erlang/tools/rebar3
user@AIM-VM MINGW64 /c/erlang/tools/rebar3 (master)
$ mkdir $USERPROFILE\\bin
user@AIM-VM MINGW64 /c/erlang/tools/rebar3 (master)
$ cp rebar3 $USERPROFILE\\bin
user@AIM-VM MINGW64 /c/erlang/tools/rebar3 (master)
$ cp rebar3.cmd $USERPROFILE\\bin
user@AIM-VM MINGW64 /c/erlang/tools/rebar3 (master)
$
```

Figure 28 – mkdir \$USERPROFILE\\Bin directory and copy rebar3, rebar3.cmd files

7. Next, add to the user variable “PATH” if it exists, or create a new user variable “PATH” and set it to “%USERPROFILE%\\bin;” without the quotes as shown in Figure 29.

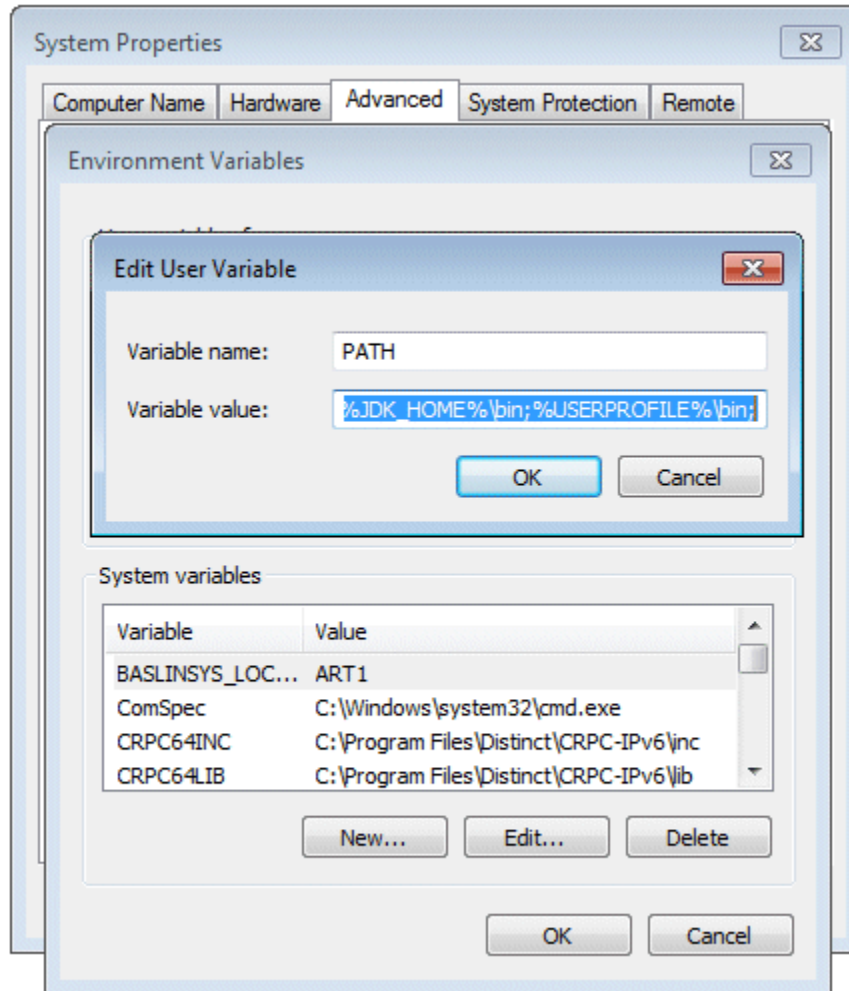


Figure 29 - Add the %USERPROFILE%\bin to user variable PATH

8. After adding the “%USERPROFILE%\bin” to the user variable path, start a new Git Bash session and verify rebar3 by performing the rebar3 version command as show in Figure 30.

```

MINGW64:/
jstarnes@JSTARNES529403 MINGW64 /
$ rebar3 version
rebar 3.13.0+build.4484.ref2612f34f on Erlang/OTP 21 Ert 10.2
jstarnes@JSTARNES529403 MINGW64 /
$

```

Figure 30 - Rebar3 Version command

9. After running the “Rebar3 version” command we need to set the REBAR_CACHE_DIR and the REBAR_GLOBAL_CONFIG_DIR. The REBAR_CACHE_DIR is used to control when HEX packages are cached by Rebar3. The REBAR_GLOBAL_CONFIG_DIR is used by Rebar3 to override usage of user’s private directory in favor of an alternate location. . If you are running on a

corporate image you may run into accessibility issues when not running on the corporate domain, which is tied to Active Directory, which sets your HOMEDRIVE H drive on network storage. Add the REBAR_CACHE_DIR and REBAR_GLOBAL_CONFIG_DIR system variables pointing to your C:\USERS\<USERNAME>, similar to Figure 31, do you'll have access when running disconnected.

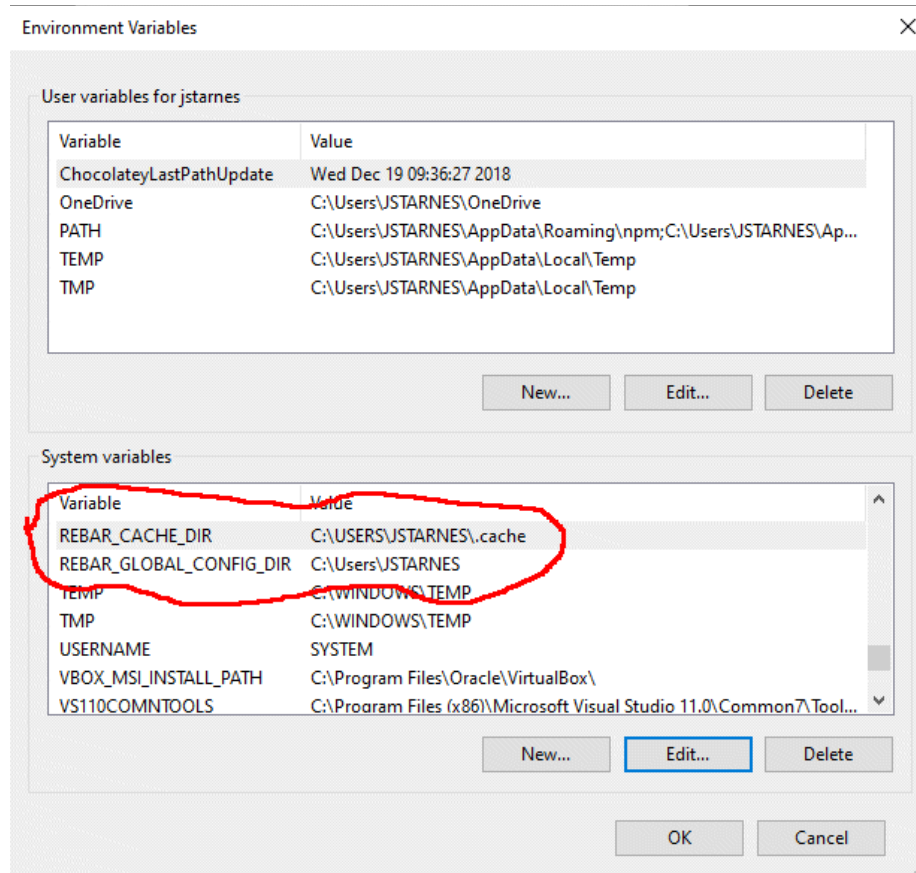


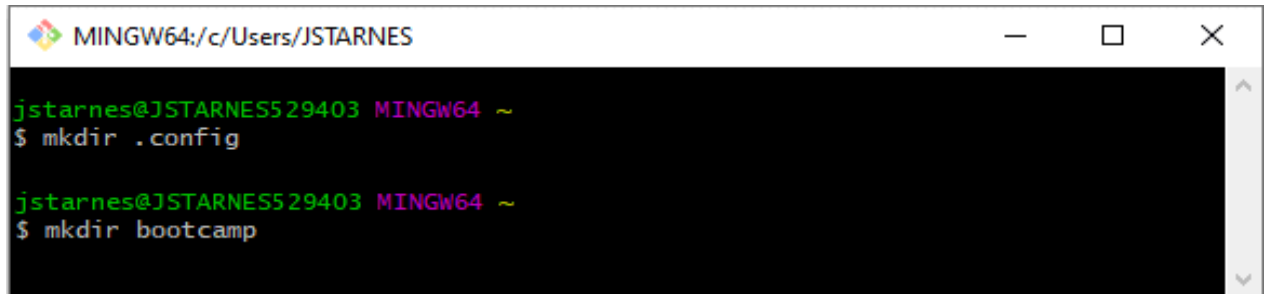
Figure 31 - REBAR_CACHE_DIR and REBAR_GLOBAL_CONFIG_DIR

- Next, in the Git Bash shell go to your default user directory, if you are running on the corporate network you may need to give the full path instead of the `cd ~` shortcut common in unix environments as show in Figure 32.



Figure 32 - Users Home Directory

- Next, in the Git Bash shell we need to create `.config` directory and `bootcamp` source directory underneath your user directory as shown in Figure 33.



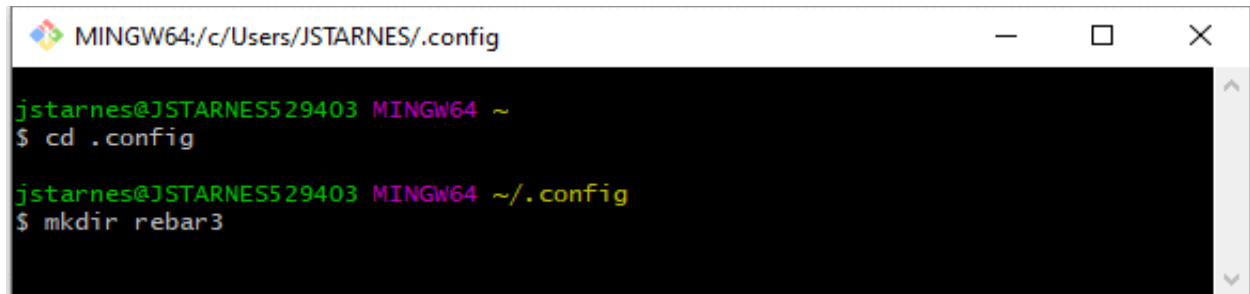
```
MINGW64:/c/Users/JSTARNES

jstarnes@JSTARNES529403 MINGW64 ~
$ mkdir .config

jstarnes@JSTARNES529403 MINGW64 ~
$ mkdir bootcamp
```

Figure 33 - mkdir .config and bootcamp

13. Next, in the Git Bash shell we need to create rebar3 directory underneath the .config directory as shown in Figure 34.



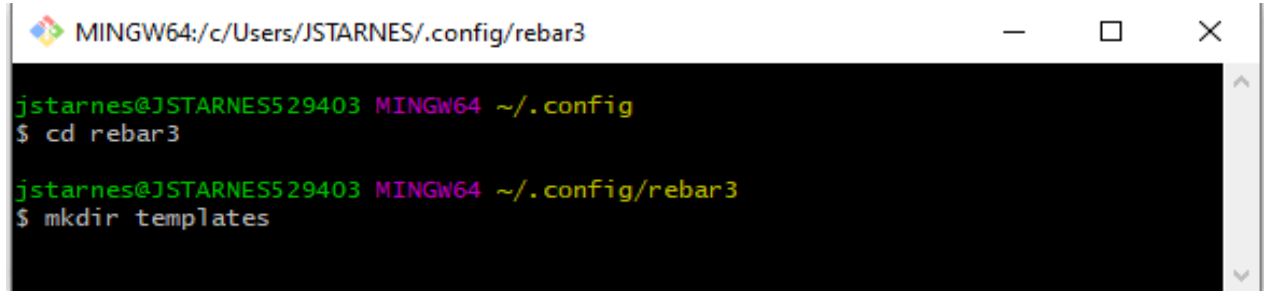
```
MINGW64:/c/Users/JSTARNES/.config

jstarnes@JSTARNES529403 MINGW64 ~
$ cd .config

jstarnes@JSTARNES529403 MINGW64 ~/.config
$ mkdir rebar3
```

Figure 34 - mkdir rebar3

14. Next, in the Git Bash shell we need to create templates directory underneath the .rebar3 directory as shown in Figure 35.



```
MINGW64:/c/Users/JSTARNES/.config/rebar3

jstarnes@JSTARNES529403 MINGW64 ~/.config
$ cd rebar3

jstarnes@JSTARNES529403 MINGW64 ~/.config/rebar3
$ mkdir templates
```

Figure 35 - mkdir templates

- Next, you'll need to copy the rebar3_templates.zip archive located in G: drive in directory "G:\STES\Jeff\.config\rebar3\templates\chp01" to your local "~/.config/rebar3/templates" directory and extract all the files the archive in templates sub folder. Once you have completed the above, in Git Bash shell go to that directory and perform the "ls -l" command to list the contents of the folder as show in Figure 36.

```

MINGW64:/c/Users/JSTARNES/.config/rebar3/templates
jstarnes@JSTARNES529403 MINGW64 /
$ cd ~/.config/rebar3/templates

jstarnes@JSTARNES529403 MINGW64 ~/.config/rebar3/templates
$ ls -l
total 115
-rw-r--r-- 1 jstarnes 1049089 388 Dec 31 15:15 app.erl
-rw-r--r-- 1 jstarnes 1049089 572 Dec 31 15:15 app.template
-rw-r--r-- 1 jstarnes 1049089 118 Dec 31 15:15 app_rebar.config
-rw-r--r-- 1 jstarnes 1049089 110 Dec 31 15:15 cmake.template
-rw-r--r-- 1 jstarnes 1049089 49 Jan 12 13:23 command.bat
-rw-r--r-- 1 jstarnes 1049089 4837 Dec 17 15:02 commonest.erl
-rw-r--r-- 1 jstarnes 1049089 291 Dec 17 15:05 commonest.template
-rw-r--r-- 1 jstarnes 1049089 552 Dec 31 15:15 escript.template
-rw-r--r-- 1 jstarnes 1049089 491 Dec 31 15:15 escript_mod.erl
-rw-r--r-- 1 jstarnes 1049089 121 Dec 31 15:15 escript_README.md
-rw-r--r-- 1 jstarnes 1049089 273 Dec 31 15:15 escript_rebar.config
-rw-r--r-- 1 jstarnes 1049089 7216 Dec 17 14:53 gen_event.erl
-rw-r--r-- 1 jstarnes 1049089 290 Dec 17 14:54 gen_event.template
-rw-r--r-- 1 jstarnes 1049089 7073 Dec 17 13:39 gen_server.erl
-rw-r--r-- 1 jstarnes 1049089 293 Dec 17 13:39 gen_server.template
-rw-r--r-- 1 jstarnes 1049089 6788 Dec 17 14:32 gen_statem.erl
-rw-r--r-- 1 jstarnes 1049089 293 Dec 17 14:40 gen_statem.template
-rw-r--r-- 1 jstarnes 1049089 156 Dec 31 15:15 gitignore
-rw-r--r-- 1 jstarnes 1049089 533 Dec 31 15:15 lib.template
-rw-r--r-- 1 jstarnes 1049089 10987 Dec 31 15:15 LICENSE
-rw-r--r-- 1 jstarnes 1049089 2467 Dec 31 15:15 Makefile
-rw-r--r-- 1 jstarnes 1049089 36 Dec 31 15:15 mod.erl
-rw-r--r-- 1 jstarnes 1049089 274 Dec 31 15:15 otp_app.app.src
-rw-r--r-- 1 jstarnes 1049089 231 Dec 31 15:15 otp_lib.app.src
-rw-r--r-- 1 jstarnes 1049089 180 Dec 31 15:15 plugin.erl
-rw-r--r-- 1 jstarnes 1049089 578 Dec 31 15:15 plugin.template
-rw-r--r-- 1 jstarnes 1049089 411 Dec 31 15:15 plugin_README.md
-rw-r--r-- 1 jstarnes 1049089 1225 Dec 31 15:15 provider.erl
-rw-r--r-- 1 jstarnes 1049089 69 Dec 31 15:15 README.md
-rw-r--r-- 1 jstarnes 1049089 97 Dec 17 14:31 readme.txt
-rw-r--r-- 1 jstarnes 1049089 38 Dec 31 15:15 rebar.config
-rw-r--r-- 1 jstarnes 1049089 876 Jan 12 13:26 release.template
-rw-r--r-- 1 jstarnes 1049089 454 Jan 1 14:55 relx_rebar.config
-rw-r--r-- 1 jstarnes 1049089 1149 Dec 31 15:15 sup.erl
-rw-r--r-- 1 jstarnes 1049089 38 Dec 31 15:15 sys.config
-rw-r--r-- 1 jstarnes 1049089 847 Dec 31 15:15 umbrella.template
-rw-r--r-- 1 jstarnes 1049089 53 Jan 12 15:53 vm.args
-rw-r--r-- 1 jstarnes 1049089 20903 Jan 1 16:56 vm.zip

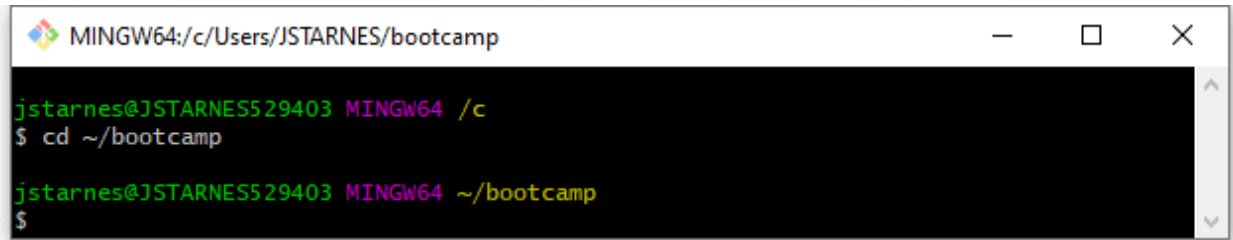
jstarnes@JSTARNES529403 MINGW64 ~/.config/rebar3/templates
$

```

Figure 36 - Templates listing

CHAPTER 1 – STARTING A PROJECT

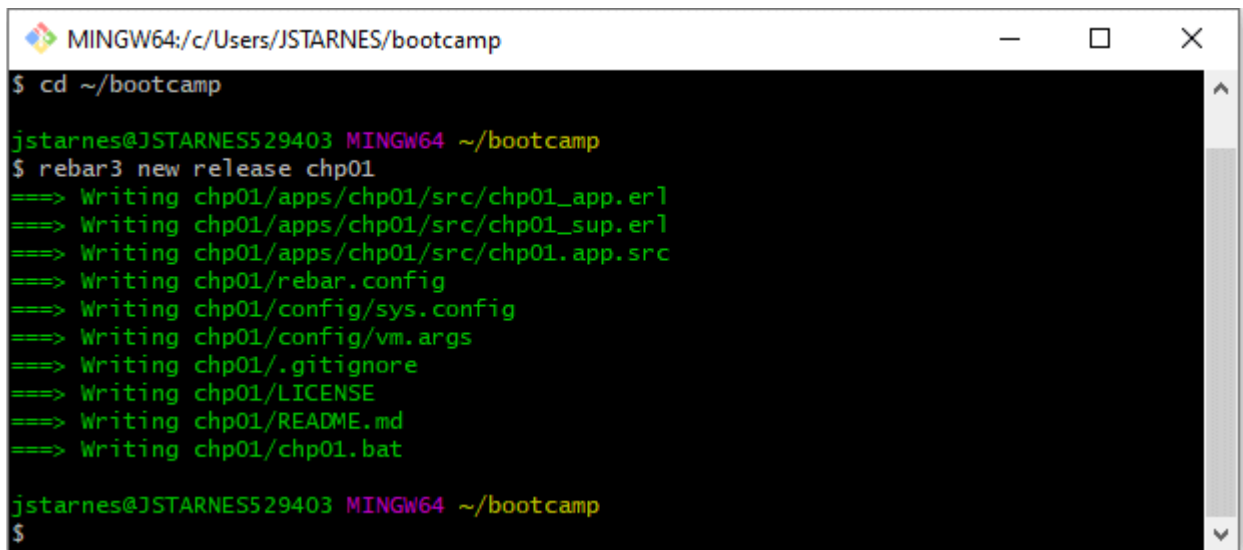
1. In Chapter 1 we are going to build our first Erlang OTP application, our equivalent to the obligatory *Hello World* application. Start a new Git Bash shell and change directory to your Bootcamp source folder as show in Figure 37. *Note, if you are using your corporate issued laptop you will want to run your Git Bash shell in elevated mode, aka as System Administrator.*



```
MINGW64:/c:/Users/JSTARNES/bootcamp
jstarnes@JSTARNES529403 MINGW64 /c
$ cd ~/bootcamp
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp
$
```

Figure 37 - cd ~/bootcamp

2. Next, without further ado enter the following command “rebar3 new release chp01” and press return to create our first Erlang OTP application as shown in Figure 38.



```
MINGW64:/c:/Users/JSTARNES/bootcamp
$ cd ~/bootcamp
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp
$ rebar3 new release chp01
==> Writing chp01/apps/chp01/src/chp01_app.erl
==> Writing chp01/apps/chp01/src/chp01_sup.erl
==> Writing chp01/apps/chp01/src/chp01.app.src
==> Writing chp01/rebar.config
==> Writing chp01/config/sys.config
==> Writing chp01/config/vm.args
==> Writing chp01/.gitignore
==> Writing chp01/LICENSE
==> Writing chp01/README.md
==> Writing chp01/chp01.bat
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp
$
```

Figure 38 - rebar3 new release chp01

3. The rebar3 new release command generated ten (10) files in total of which five (5) are in the chp01 folder, three (3) are in the apps/chp01/src sub folder, and two (2) are in the config sub folder.

4. Next, change directory to the newly created chp01 folder, this folder is the root folder for the chp01 application and is where you configure your application top level Dependencies, Profiles, Plugins, etc. It is also the primary directory for compilation and builds for your application. Let's take a look at the rebar3.config, config/sys.config and config/vm.args files as show in Listing 1

Listing 1 – rebar3.config, config/sys.config, and config/vm.args files

```
$ cd ./chp01
$ cat rebar3.config

{erl_opts, [debug_info]}.
{deps, []}.

{relx, [{release, {chp01, "0.1.0"},
                  [chp01,
                   sasl]},

        {sys_config, "./config/sys.config"},
        {vm_args, "./config/vm.args"},

        {dev_mode, false},
        {include_erts, false},
        {system_libs, false},
        {extended_start_script, true}]
}.

{profiles, [{prod, [{relx, [{dev_mode, false},
                           {include_erts, true},
                           {system_libs, true},
                           {include_src, false},
                           {debug_info, strip}]}]}]}
}.

$ cat config/sys.config

[
  {chp01, []}
].

$ cat config/vm.args

-sname chp01

-setcookie 8812

+K true
+A30
```

- Next, let's visit the apps/chp01/src sub folder and review the three source files chp01.app.src, chp01_app.erl, and chp01_sup.erl. The chp0.app.src file defines the application name, description, vsn, modules, registered, and applications. The following link <https://erlang.org/doc/man/app.html> gives more detail on the meaning of these fields and we will be working with them in future chapters.

Listing 2 – chp01.app.src

```
$ cat chp01.app.src

application, chp01,
[{description, "An OTP application"},
 {vsn, "0.1.0"},
 {registered, []},
 {mod, {chp01_app, []}},
 {applications,
  [kernel,
   stdlib
  ]},
 {env, []},
 {modules, []},

 {licenses, ["Apache 2.0"]},
 {links, []}
].
```

The chp0_app.erl file is the Erlang source file for the application callback file for the chp01 application. The `-behavior(application).` tag defines this source supports the application callback functions `start/2` and `stop/1`, whereas the `/2` means the function has an arity (number of arguments). The following link https://erlang.org/doc/design_principles/applications.html#application-concept gives more detail on the usage of the application behavior. Behaviors are a fundamental part of Erlang OTP and we will be learning more about them in future chapters.

Listing 3 - chp01_app.erl

```
$ cat chp01_app.erl

%%%-----
%% @doc chp01 public API
%% @end
%%%-----

-module(chp01_app).

-behaviour(application).

-export([start/2, stop/1]).

start(_StartType, _StartArgs) ->
    chp01_sup:start_link().

stop(_State) ->
    ok.

%% internal functions
```

The `chp0_sup.erl` file is the Erlang source file for the supervisor callback file for the `chp01` application. The `-behavior(supervisor).` tag defines this source supports the application callback functions `start/0` and `init/1`. The following link https://erlang.org/doc/design_principles/sup_princ.html gives more detail on the usage of the supervisor behavior. The Supervisor plays a critical role in providing a fault tolerant solution in Erlang OTP applications. We will be learning more about Supervisors and the process tree in future chapters.

Listing 4 - `chp01_sup.erl`

```
$ cat chp01_sup.erl

%%%-----
%% @doc chp01 top level supervisor.
%% @end
%%%-----

-module(chp01_sup).

-behaviour(supervisor).

-export([start_link/0]).

-export([init/1]).

-define(SERVER, ?MODULE).

start_link() ->
    supervisor:start_link({local, ?SERVER}, ?MODULE, []).

%% sup_flags() = #{strategy => strategy(),           % optional
%%               intensity => non_neg_integer(), % optional
%%               period => pos_integer()}           % optional
%% child_spec() = #{id => child_id(),                % mandatory
%%                 start => mfargs(),                % mandatory
%%                 restart => restart(),             % optional
%%                 shutdown => shutdown(),           % optional
%%                 type => worker(),                 % optional
%%                 modules => modules()}             % optional
init([]) ->
    SupFlags = #{strategy => one_for_all,
                 intensity => 0,
                 period => 1},
    ChildSpecs = [],
    {ok, {SupFlags, ChildSpecs}}.

%% internal functions
```

6. It wouldn't be considered a Hello World application without the obligatory output of "Hello World". So, let's edit the `chp01_sup.erl` `init([])` function header the following line without the quotes `"io:format("Hello World!~n", [])`, your file should be similar to Listing 5.

Listing 5 - Updated `chp01_sup.erl`

```
%%%-----
%% @doc chp01 top level supervisor.
%% @end
%%%-----

-module(chp01_sup).

-behaviour(supervisor).

-export([start_link/0]).

-export([init/1]).

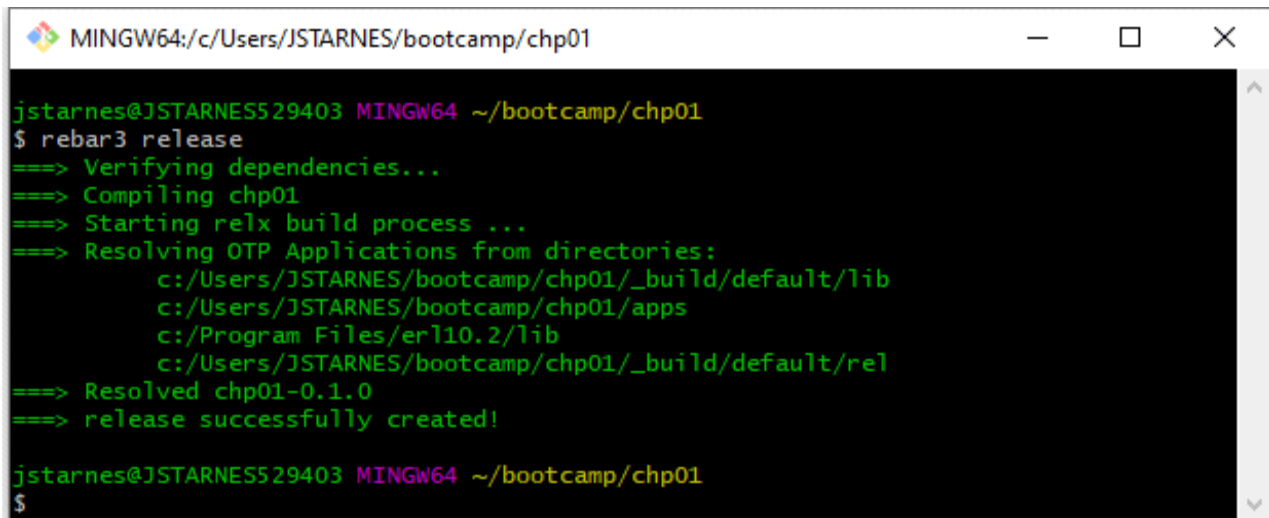
-define(SERVER, ?MODULE).

start_link() ->
    supervisor:start_link({local, ?SERVER}, ?MODULE, []).

%% sup_flags() = #{strategy => strategy(),           % optional
%%               intensity => non_neg_integer(), % optional
%%               period => pos_integer()}           % optional
%% child_spec() = #{id => child_id(),                % mandatory
%%                  start => mfargs(),                % mandatory
%%                  restart => restart(),             % optional
%%                  shutdown => shutdown(),           % optional
%%                  type => worker(),                 % optional
%%                  modules => modules()}             % optional
init([]) ->
    io:format("Hello World!~n", []),
    SupFlags = #{strategy => one_for_all,
                  intensity => 0,
                  period => 1},
    ChildSpecs = [],
    {ok, {SupFlags, ChildSpecs}}.

%% internal functions
```

7. Next, we are final ready to final build our application and run it. Before building the final build, enter in the following command to return to the top build folder `"cd ~/bootcamp/chp01"`. Enter the following Rebar3 command to build a release as shown in Figure 39. Note that this is known as a development release and later we will discuss how the production release is produced.



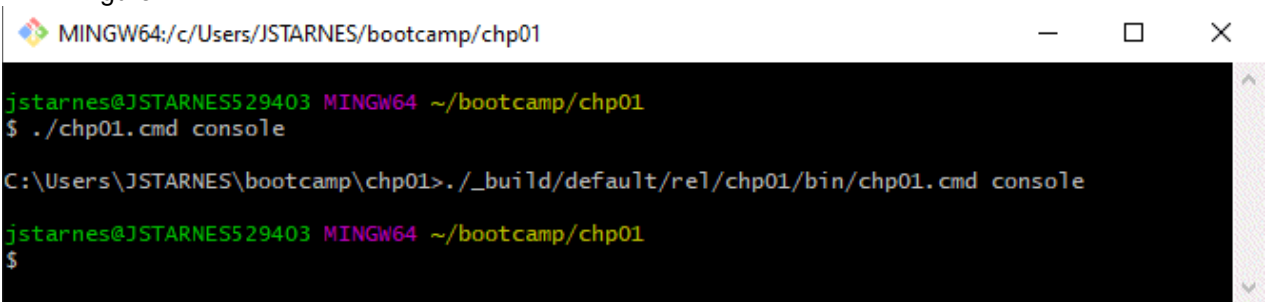
```
MINGW64:/c:/Users/JSTARNES/bootcamp/chp01

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$ rebar3 release
==> Verifying dependencies...
==> Compiling chp01
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
      c:/Users/JSTARNES/bootcamp/chp01/_build/default/lib
      c:/Users/JSTARNES/bootcamp/chp01/apps
      c:/Program Files/erl10.2/lib
      c:/Users/JSTARNES/bootcamp/chp01/_build/default/re1
==> Resolved chp01-0.1.0
==> release successfully created!

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$
```

Figure 39 – Rebar3 release

- Next, let's run our application by entering “./chp01.cmd console” on the command line in the “~/bootstrap/chp01” folder as show in Figure 40. The output from our chp01 application can be seen in Figure 41.



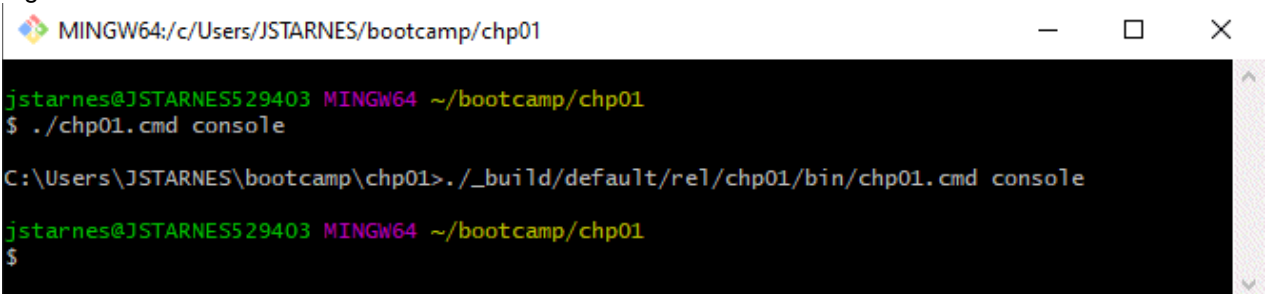
```
MINGW64:/c:/Users/JSTARNES/bootcamp/chp01

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$ ./chp01.cmd console

C:\Users\JSTARNES\bootcamp\chp01>._build/default/re1/chp01/bin/chp01.cmd console

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$
```

Figure 40



```
MINGW64:/c:/Users/JSTARNES/bootcamp/chp01

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$ ./chp01.cmd console

C:\Users\JSTARNES\bootcamp\chp01>._build/default/re1/chp01/bin/chp01.cmd console

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$
```

Figure 40 – chp01.cmd console command

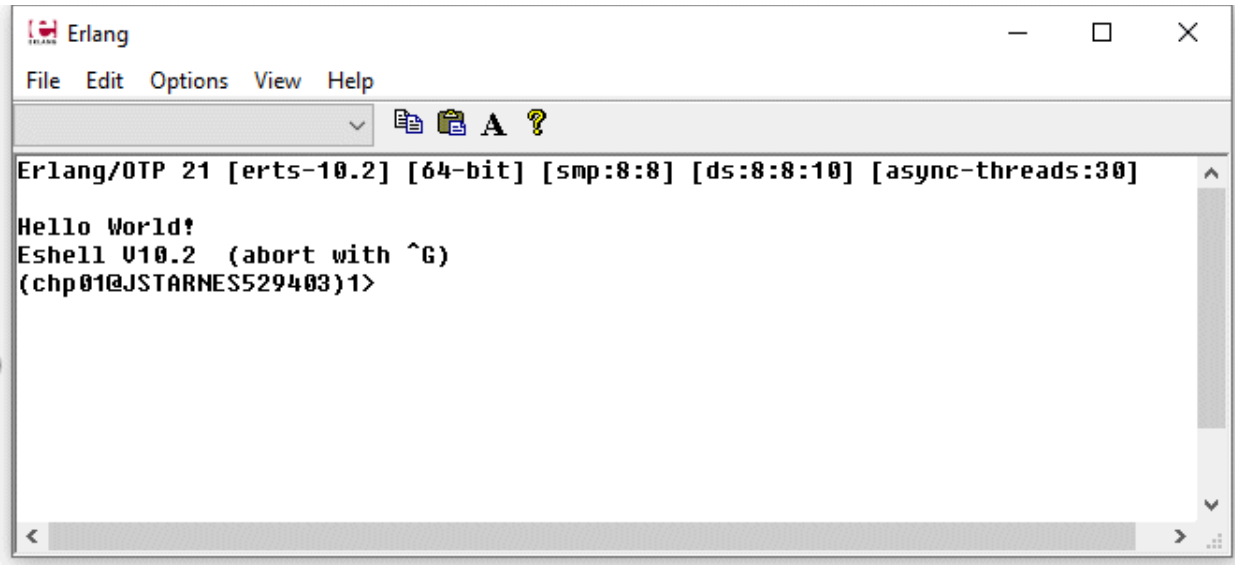
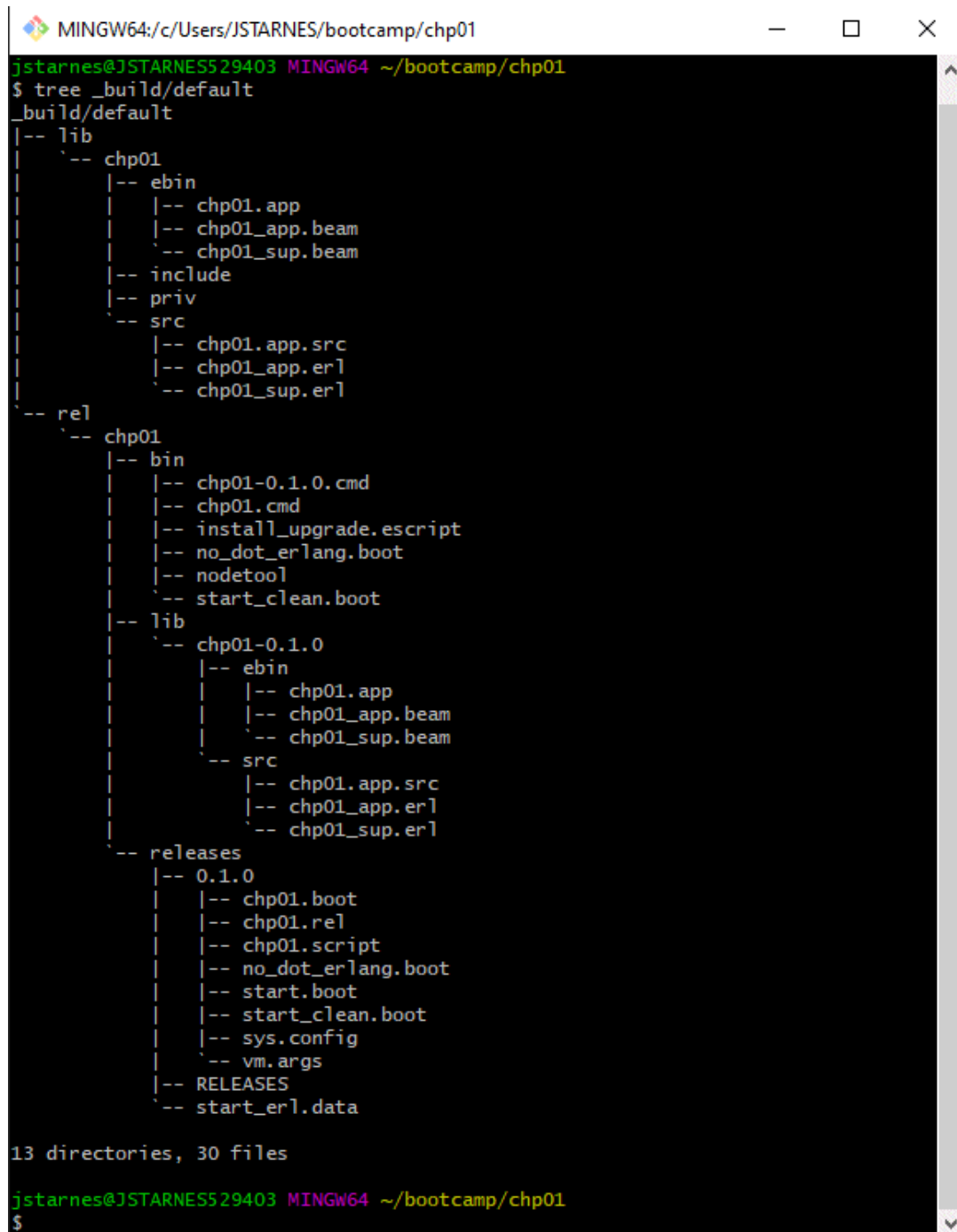


Figure 41 - chp01 console window

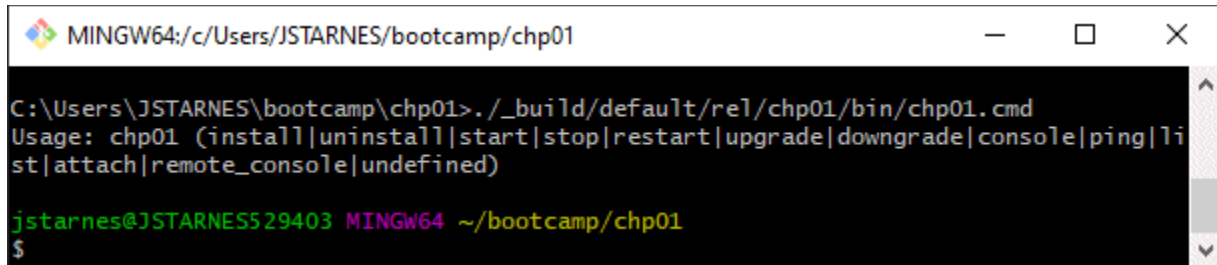
9. Before going to the next step, exit the chp01 console window by pressing the [X] button on the top right of window.
10. Before we delve into the available commands in the generated chp01.cmd command file, let's explore the _build/default folder by using the gnu tree program ([gnuwin32/tree-1.5.2.2-bin.zip](#)) that I copied into the C:\Program Files\Git\usr\bin folder and get a feel of what is being generated by rebar3 release command as shown in Figure 42.



```
MINGW64:/c/Users/JSTARNES/bootcamp/chp01
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$ tree _build/default
_build/default
|-- lib
|   |-- chp01
|   |   |-- ebin
|   |   |   |-- chp01.app
|   |   |   |-- chp01_app.beam
|   |   |   |-- chp01_sup.beam
|   |   |-- include
|   |   |-- priv
|   |   |-- src
|   |       |-- chp01.app.src
|   |       |-- chp01_app.erl
|   |       |-- chp01_sup.erl
|   |-- rel
|   |   |-- chp01
|   |   |   |-- bin
|   |   |   |   |-- chp01-0.1.0.cmd
|   |   |   |   |-- chp01.cmd
|   |   |   |   |-- install_upgrade.escript
|   |   |   |   |-- no_dot_erlang.boot
|   |   |   |   |-- nodetool
|   |   |   |   |-- start_clean.boot
|   |   |   |-- lib
|   |   |       |-- chp01-0.1.0
|   |   |       |   |-- ebin
|   |   |       |   |   |-- chp01.app
|   |   |       |   |   |-- chp01_app.beam
|   |   |       |   |   |-- chp01_sup.beam
|   |   |       |-- src
|   |   |           |-- chp01.app.src
|   |   |           |-- chp01_app.erl
|   |   |           |-- chp01_sup.erl
|   |   |-- releases
|   |       |-- 0.1.0
|   |       |   |-- chp01.boot
|   |       |   |-- chp01.rel
|   |       |   |-- chp01.script
|   |       |   |-- no_dot_erlang.boot
|   |       |   |-- start.boot
|   |       |   |-- start_clean.boot
|   |       |   |-- sys.config
|   |       |   |-- vm.args
|   |       |-- RELEASES
|   |       |-- start_erl.data
13 directories, 30 files
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$
```

Figure 42 - rebar3 release output

11. In step #8 we ran the chp01.cmd file with a console argument pass into the chp01.cmd file generated by running the rebar3 release command. Let's run the command again, but this time without any argument and see what happens as show in Figure 43.



```
MINGW64:/c/Users/JSTARNES/bootcamp/chp01
C:\Users\JSTARNES\bootcamp\chp01> ./_build/default/rel/chp01/bin/chp01.cmd
Usage: chp01 (install|uninstall|start|stop|restart|upgrade|downgrade|console|ping|list|attach|remote_console|undefined)

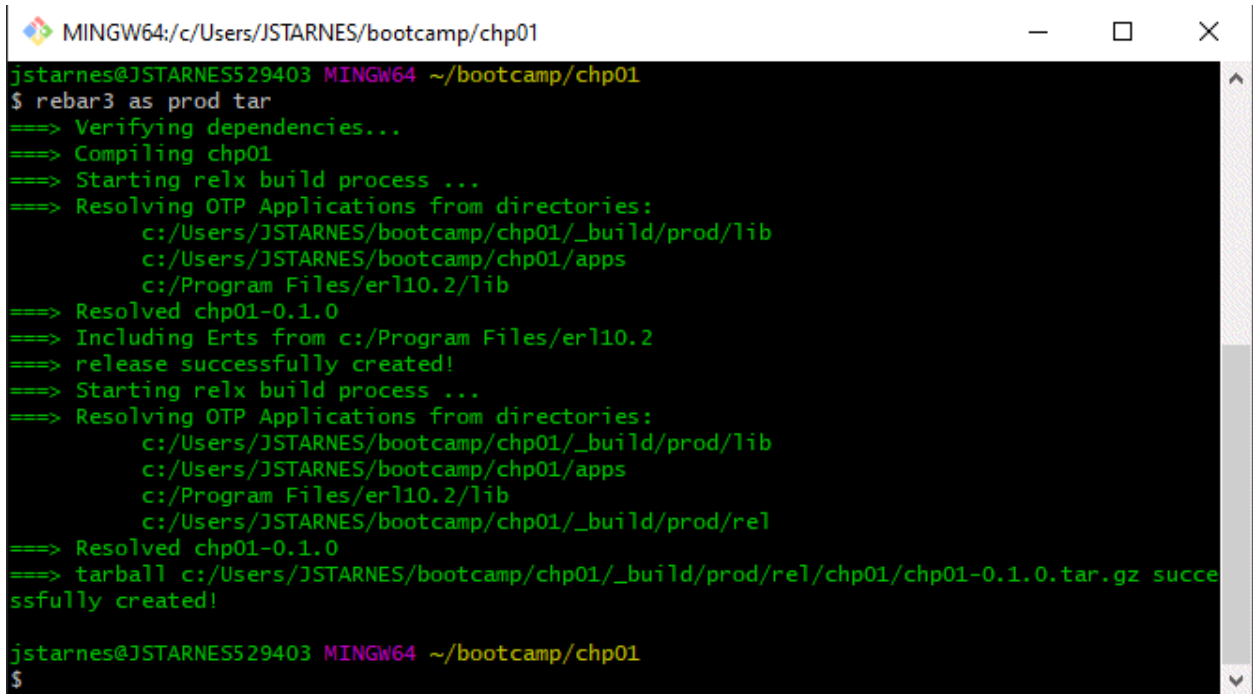
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$
```

Figure 43 - chp01.bat no arguments

12. The chp01.cmd file returns a Usage statement and a list of available options that are applicable to the chp01 application as shown below:

- console – Runs the Erlang OTP application as a process with a console.
- *install* – Installs the Erlang OTP application to run as a Windows Service
- uninstall – Uninstalls the installed Erlang OTP application that is running as a Windows Service
- start – Starts the Erlang OTP application that is running as a Windows Service
- stop – Stops the Erlang OTP application that is running as a Windows Service
- restart – Restarts the Erlang OTP application that is running as a Windows Service
- upgrade – Upgrades a currently running Erlang OTP application release to a supplied version
- downgrade – Downgrades a currently running Erlang OTP application release to a supplied version
- console – Runs the Erlang OTP application inside a window application
- ping – Pings the currently running Erlang OTP application regardless of it is a Windows Service or just a Windows application.
- list – Lists installed Erlang Services
- attach/remote_console – Attaches to a currently running Erlang OTP application

13. So far our exercises we have been strictly in development mode which out of the above list of options we are limited to just console, ping, attach/remote_console. Let's go to the next step and generate a production release of our chp01 application by entering the following command shown in Figure 44.



```
MINGW64: c:/Users/JSTARNES/bootcamp/chp01
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$ rebar3 as prod tar
==> Verifying dependencies...
==> Compiling chp01
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
      c:/Users/JSTARNES/bootcamp/chp01/_build/prod/lib
      c:/Users/JSTARNES/bootcamp/chp01/apps
      c:/Program Files/erl10.2/lib
==> Resolved chp01-0.1.0
==> Including Erts from c:/Program Files/erl10.2
==> release successfully created!
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
      c:/Users/JSTARNES/bootcamp/chp01/_build/prod/lib
      c:/Users/JSTARNES/bootcamp/chp01/apps
      c:/Program Files/erl10.2/lib
      c:/Users/JSTARNES/bootcamp/chp01/_build/prod/rel
==> Resolved chp01-0.1.0
==> tarball c:/Users/JSTARNES/bootcamp/chp01/_build/prod/rel/chp01/chp01-0.1.0.tar.gz successfully created!

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$
```

Figure 44 – rebar3 as prod tar

14. The rebar3 as prod tar command generates the chp01-0.1.0.tar.gz file that is used for installation on a target machine does not have Erlang installed on it. The Erlang application releases fall into two separate categories, the first and most common is the installation on a target machine without Erlang installed, the second option is installation on a machine that has Erlang already installed. The latter case will not be covered by our Erlang Bootcamp as it is more involved and unfortunately not very well documented.
15. Similar to step #9 before, it's a good idea to examine the contents of the _build/prod folder that is generated by the rebar3 as prod tar command. However, I'm going to leave that exercise up to you as the content is significantly larger as it contains supporting Erlang kernel and system libraries necessary for installation on a computer which does not have Erlang installed.

16. Next, let's create a myrel folder underneath chp01 folder to install our production release and copy the generated chp01-0.1.0.tar.gz into that directory and then run the tar -zxvf command to extract the release archive as shown in Figure 45.

```

MINGW64: c:/Users/JSTARNES/bootcamp/chp01/myrel

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$ mkdir myrel

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$ cp _build/prod/rel/chp01/chp01-0.1.0.tar.gz myrel

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01
$ cd myrel

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01/myrel
$ tar -zxvf chp01-0.1.0.tar.gz

```

Figure 45 – Installation of chp01 production release via tar -zxvf chp01-0.1.0.tar.gz

17. Next, let's install the chp01 application and use some of the chp01.cmd commands to get a feel for how they work and their potential uses cases. Run the following steps: a thru h list below:

- a. Run the bin/chp01.cmd console command
- b. Run the bin/chp01.cmd install command

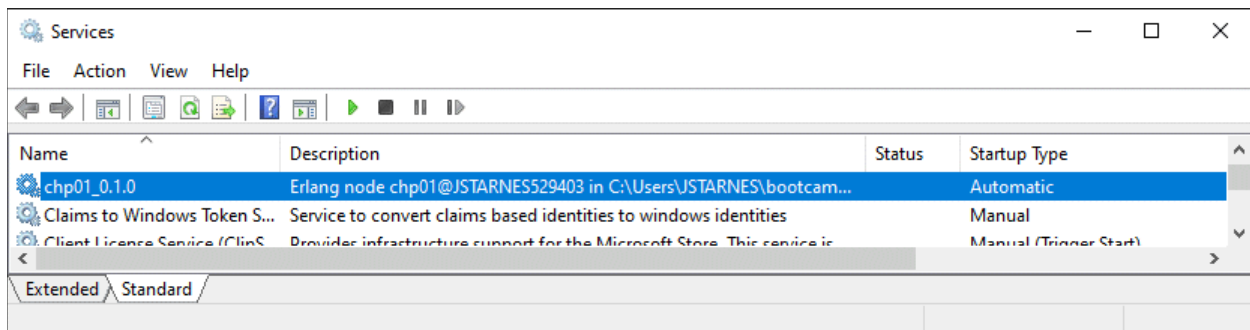


Figure 46 – bin/chp01.cmd installed as Windows Service

- c. Run the bin/chp01.cmd list command

Listing 6 – bin/chp01.cmd list command

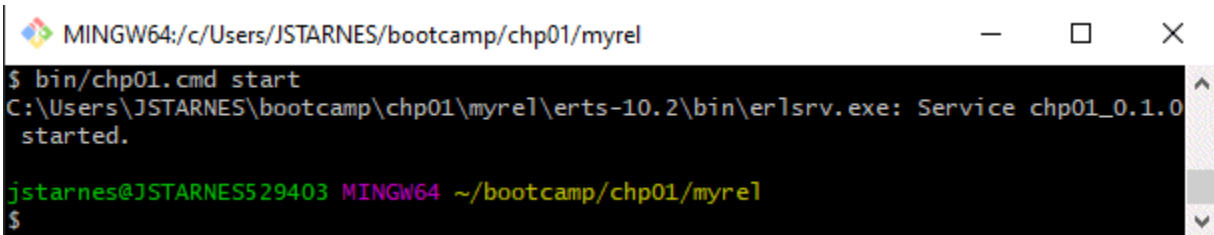
```

$ bin/chp01.cmd list

Service name: chp01_0.1.0
StopAction: init:stop().
OnFail: ignore
Machine: C:\Users\JSTARNES\bootcamp\chp01\myrel\erts-10.2\bin\start_erl.exe
WorkDir: C:\Users\JSTARNES\bootcamp\chp01\myrel
SName: chp01@JSTARNES529403
Priority: default
DebugType: none
Args: +K true +A30 -setcookie 8812 ++ -rootdir
"C:\Users\JSTARNES\bootcamp\chp01\myrel "
Internal ServiceName: chp0101d5cfbd40e6e520
Comment: Erlang node chp01@JSTARNES529403 in C:\Users\JSTARNES\bootcamp\chp01\myrel
Env:

```

- d. Run the bin/chp01.cmd start command

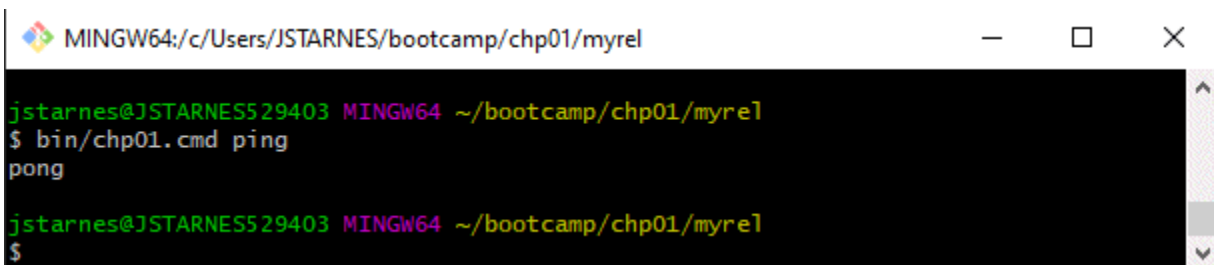


```
MINGW64:/c/Users/JSTARNES/bootcamp/chp01/myrel
$ bin/chp01.cmd start
C:\Users\JSTARNES\bootcamp\chp01\myrel\erts-10.2\bin\erlsrv.exe: Service chp01_0.1.0
started.

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01/myrel
$
```

Figure 47 – bin/chp01.cmd start command

- e. Run the bin/chp01.cmd ping command



```
MINGW64:/c/Users/JSTARNES/bootcamp/chp01/myrel
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01/myrel
$ bin/chp01.cmd ping
pong

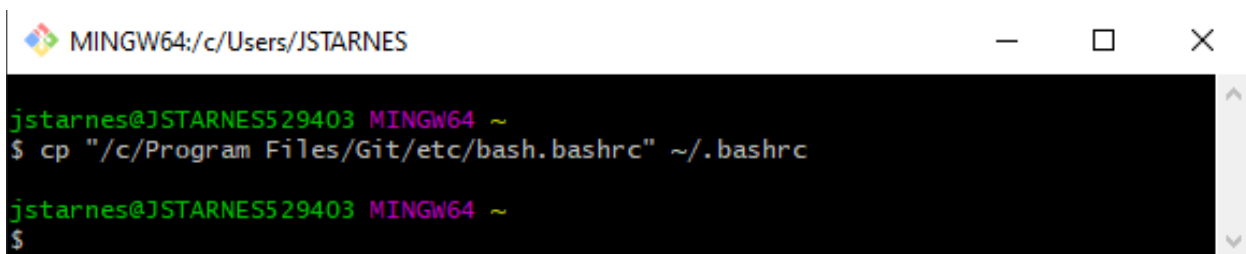
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01/myrel
$
```

Figure 48 - bin/chp01.cmd ping command

Note, if you are running on a corporate laptop disconnected from the domain you may get an error message running the above command as it writes a temporary .erlang.cookie file in your H: drive. To get around this issue set your \$HOMEDRIVE to \$USERPROFILE value via the following command:

```
$ export HOMEDRIVE=$USERPROFILE
```

To make the above more permanent you can copy the “C:\Program Files\Git\etc\bash.bashrc” file to your user drive and ~/.bashrc shown in Figure 49 and add the export HOMEDRIVE=\$USERPROFILE to the very end of the file.

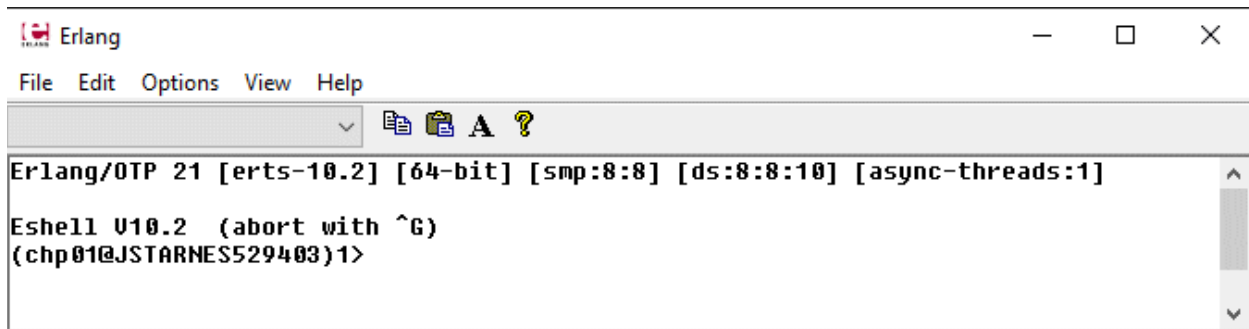


```
MINGW64:/c/Users/JSTARNES
jstarnes@JSTARNES529403 MINGW64 ~
$ cp "/c/Program Files/Git/etc/bash.bashrc" ~/.bashrc

jstarnes@JSTARNES529403 MINGW64 ~
$
```

Figure 49 - copy bash.bashrc to ~/.bashrc

- f. Run the bin/chp01.cmd attach command



```

Erlang
File Edit Options View Help
Erlang/OTP 21 [erts-10.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1]
Eshell U10.2 (abort with ^G)
(chp01@JSTARNES529403)1>

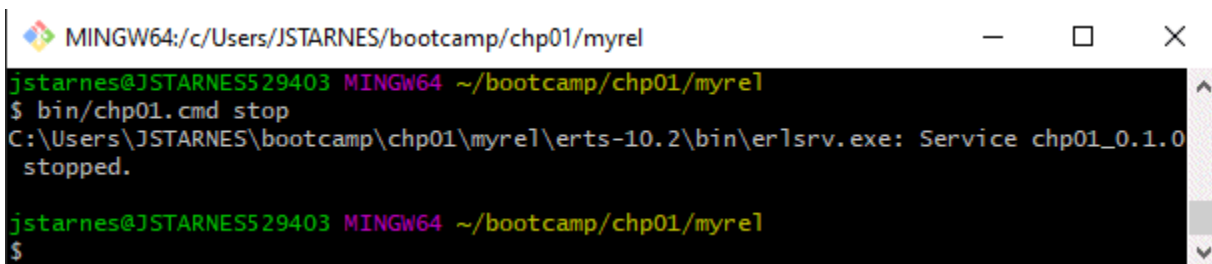
```

Figure 50 - bin/chp01.cmd attach command

The attach/remote_console command is a very significant feature as part of the Erlang OTP application as it provides a mechanism for communicating to the running application and run commands, etc. This is a must have feature for management of operational solutions without stopping the application. We will be demonstrating the power of this feature for creating our own management functions as well as running utilities such as tracing, etc. in future chapters.

*** Important *** The remote console application runs in the same context as the application itself, in order to safely exit your running application, press the CTRL+G keys followed by q character and return character. Otherwise, you can actually kill your application by clicking on the [X] button. We will discuss the capabilities of the remote console in a future chapter.

- g. Run the bin/chp01.cmd stop command



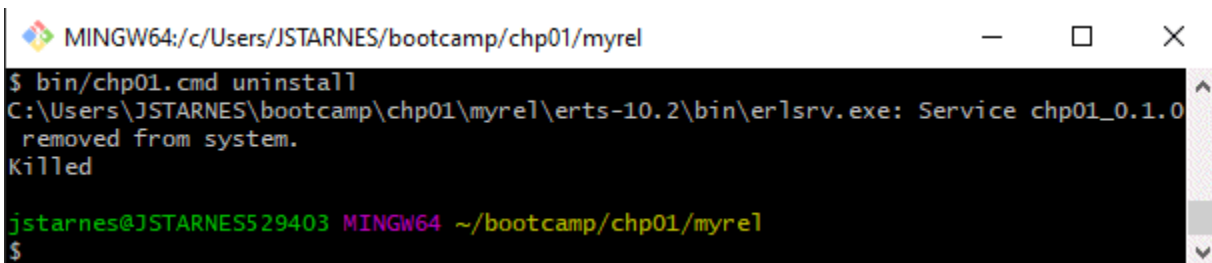
```

MINGW64:/c/Users/JSTARNES/bootcamp/chp01/myrel
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01/myrel
$ bin/chp01.cmd stop
C:\Users\JSTARNES\bootcamp\chp01\myrel\erts-10.2\bin\erlsrv.exe: Service chp01_0.1.0
stopped.
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01/myrel
$

```

Figure 51 - bin/chp01.cmd command

- h. Run the bin/chp01.cmd uninstall command



```

MINGW64:/c/Users/JSTARNES/bootcamp/chp01/myrel
$ bin/chp01.cmd uninstall
C:\Users\JSTARNES\bootcamp\chp01\myrel\erts-10.2\bin\erlsrv.exe: Service chp01_0.1.0
removed from system.
Killed
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp01/myrel
$

```

Figure 52 - bin/chp01 uninstall command

18. Before we leave the discussion on the builtin set of commands that allows you to manage your release, it is important to note that sometimes your application may have a need for additional commands. Many Erlang

applications like [RabbitMQ](#) and/or [Riak](#) provide additional command line administrative scripts and/or /programs that extend the basic set of commands listed. We will be covering our own extensions in a future chapter.

Summary

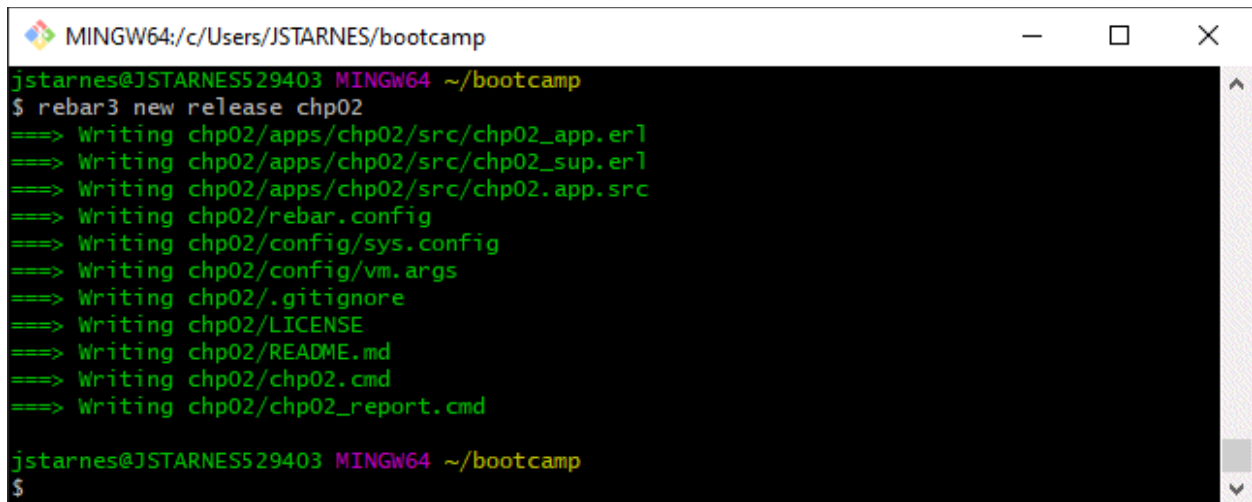
We covered significant ground in in the Erlang Bootcamp, we accomplished the follow items:

- Starting a Project learned the *rebar3 new release <app_name>* command
- Reviewed generated config files: *rebar3.config*, *config/sys.config*, and *config/vm.args*
- Reviewed generated source files: *chp01.app.src*, *chp01_app.src*, and *chp01_sup.erl*
- Modified source file: *chp01_sup.erl* to add “Hello World” output.
- Built a development release using the *rebar3 release* command
- Ran the *chp01.cmd console* command
- Examined the directory structure of the *_build/default* as a result of the output of the development release
- Reviewed the *chp01.cmd* options
- Built our first production release and installed it in our *myrel* folder
- Examined the directory structure of the *_build/prod* as a result of the output of the production release
- Ran the *bin/chp01.cmd* command to run *chp01* application as a standard alone application
- Ran the *bin/chp01.cmd* command to install the *chp01* application as a Windows Service
- Ran the *bin/chp01.cmd list* command to list the details of any Erlang Windows Services
- Ran the *bin/chp01.cmd ping* command to ping the *chp01* application
- Ran the *bin/chp01.cmd attach* command giving the operator access *chp01* application runtime
- Ran the *bin/chp01.cmd stop* command to stop a running *chp01* Windows Service
- Ran the *bin/chp01.cmd unistall* command to uninstall *chp01* Windows Service
- Learn that the above commands can be expanded for our Erlang applications like *RabbitMQ* or *Riak* Erlang applications

While we covered a lot of great information, we certainly skip over many of the details that we will need to know to be productive. We will get more involved with the generated *rebar* generated files in the following chapters, but for now we’ve at least covered the start, development, and production release of an Erlang OTP application using *rebar3*.

CHAPTER 2 - ERLANG IN PRODUCTION

1. In Chapter 2 we will learn about Erlang in Production [4] a presentation by Jesper Louis Andersen of Erlang Solutions that describes the necessary tooling for any successful Erlang application. He describes the available tools and their purpose. I recommend you take a look at the slide deck yourself, as we cut to the chase and add the necessary tooling for our hypothetical application. Note at the end of the Chapter 2, you will have the basis starting point for any conceivable Erlang OTP application.
2. The first step is to the following command “rebar3 new release chp02” and press return to create our second Erlang OTP application as shown in Figure 53.



```
MINGW64:/c/Users/JSTARNES/bootcamp
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp
$ rebar3 new release chp02
==> Writing chp02/apps/chp02/src/chp02_app.erl
==> Writing chp02/apps/chp02/src/chp02_sup.erl
==> Writing chp02/apps/chp02/src/chp02.app.src
==> Writing chp02/rebar.config
==> Writing chp02/config/sys.config
==> Writing chp02/config/vm.args
==> Writing chp02/.gitignore
==> Writing chp02/LICENSE
==> Writing chp02/README.md
==> Writing chp02/chp02.cmd
==> Writing chp02/chp02_report.cmd
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp
$
```

Figure 53 - rebar3 new release chp02

3. The first item we are going to add to chp02 application is the Erlang System Architecture Support Libraries (SASL) adds the following services: alarm_handler, release_handler, systools. Open the config/sys.config file and add the following content underneath the tuple {chp02, []} as show in Listing 7.

Listing 7 - SASL Config

```
[
  {chp02, []},
  %% kernel
  {kernel, [{logger_sasl_compatible, true}]},
  %% SASL config
  {sasl, [
    {sasl_error_logger, {file, "log/sasl-error.log"}},
    {errlog_type, error},
    {error_logger_mf_dir, "log/sasl"},           % Log directory
    {error_logger_mf_maxbytes, 10485760},       % 10 MB max file size
    {error_logger_mf_maxfiles, 7}               % 7 files max
  ]}
].
```


- Next, we want to add the `{overlay [{mkdir, "./log/sasl"}]}` command underneath the `{extended_start_script, true}` tuple to create the `./log/sasl` directories for the development and production releases as show in Listing 8.

Listing 8 - rebar.config add overlay command

```
{erl_opts, [debug_info]}.
{deps, []}.

{relx, [{release, {chp02, "0.1.0"},
                  [chp02,
                   sasl]},

        {sys_config, "./config/sys.config"},
        {vm_args, "./config/vm.args"},

        {dev_mode, false},
        {include_erts, false},
        {system_libs, false},
        {extended_start_script, true},
        {overlay, [{mkdir, "./log/sasl"}]}]}.

{profiles, [{prod, [{relx, [{dev_mode, false},
                            {include_erts, true},
                            {system_libs, true},
                            {include_src, false},
                            {debug_info, strip}]}]}]}.
}
```

- Next, we want to add the Lager dependency package to the rebar.config file which extends the existing Erlang logger functionality with log rotation and crash dump files, etc. Add the dependency package as shown in Listing 9.

Listing 9 - Lager package dependency

```
{erl_opts, [debug_info, {parse_transform, lager_transform}]}.
{deps, [
    %% Packages
    {lager, "3.6.3"}
]}.

{relx, [{release, {chp02, "0.1.0"},
                  [chp02, sasl, lager]},

        {sys_config, "./config/sys.config"},
        {vm_args, "./config/vm.args"},

        {dev_mode, false},
        {include_erts, false},
        {system_libs, false},
        {extended_start_script, true},
        {overlay, [{mkdir, "./log/sasl"}]}]}.

{profiles, [{prod, [{relx, [{dev_mode, false},
                            {include_erts, true},
                            {system_libs, true},
                            {include_src, false},
                            {debug_info, strip}]}]}]}.
}
```

6. Next, add the Lager configuration for crash.log and rotation of error.log, console.log, and debug.log as show in Listing 10. The lager library intercepts the Erlang normal logging and creates plain ascii output files and adds size, date, and count retention parameters. Refer to [User Content Internal Log Rotation](#) for more details on what the meaning of the parameters are, the \$D0, rotates every day at midnight, the count limits the number of logs by size. Note the {logger_sasl_compatible, true} tuple is set to allow the Lager library to work with the new Erlang OTP 21 logging capability.

Listing 10 - Lager config files

```
[
  {chp02, []},
  %% kernel
  {kernel, [{logger_sasl_compatible, true}]},
  %% SASL config
  {sasl, [
    {sasl_error_logger, {file, "log/sasl-error.log"}},
    {errlog_type, error},
    {error_logger_mf_dir, "log/sasl"},           % Log directory
    {error_logger_mf_maxbytes, 10485760},       % 10 MB max file size
    {error_logger_mf_maxfiles, 7}               % 7 files max
  ]},
  %% Lager config - [debug, info, notice, warning, error, critical, alert, emergency, none]
  {lager, [
    {log_root, "log"},
    {crash_log, "crash.log"},
    {crash_log_msg_size, 65536},
    {crash_log_size, 10485760},
    {crash_log_date, "$D0"},
    {crash_log_count, 7},
    {handlers, [
      {lager_file_backend,
        [{file, "error.log"}, {level, error}, {size, 10485760}, {date, "$D0"}, {count, 50}]},
      {lager_file_backend,
        [{file, "console.log"}, {level, info}, {size, 10485760}, {date, "$D0"}, {count, 50}]},
      {lager_file_backend,
        [{file, "debug.log"}, {level, debug}, {size, 10485760}, {date, "$D0"}, {count, 50}]}
    ]}
  ]}
].
```

The Lager library also creates a crash.log with retention parameters and gives a single location to see the really critical failures, the ones that caused a process to crash. This is really important, because while Erlang OTP applications are fault tolerant, individual processes can and do crash and the crash file gives us the details about the crash, including the state of the process, the raw data and the stack trace of the modules, functions and arguments.

Listing 11 is a sample crash file that shows a function clause error for the `eg_richText` module and `normalize_str` function which is not expecting a list of `[0, []]` instead of a string.

Listing 11 – Sample Crash File

```
2018-06-20 07:54:19 =CRASH REPORT====
crasher:
  initial_call: rp_rptpro:init_work/1
  pid: <0.8594.6314>
  registered_name: []
  exception_error:
    {function_clause, [{eg_richText, normalize_str, [0, []], [{file, "d:/SPRTVL/source/sprtv1/_build/default/lib/ngerlguten/src/eg_richText.erl"}, {line, 165}]}], [{eg_richText, str2richText, 6, [{file, "d:/SPRTVL/source/sprtv1/_build/default/lib/ngerlguten/src/eg_richText.erl"}, {line, 144}]}], [{eg_pdf, get_string_width, 3, [{file, "d:/SPRTVL/source/sprtv1/_build/default/lib/ngerlguten/src/eg_pdf.erl"}, {line, 266}]}], [{rp_rptpro, cell, 8, [{file, "d:/SPRTVL/source/sprtv1/_build/default/lib/reports/src/rp_rptpro.erl"}, {line, 1068}]}], [{rp_rptpro, station_row, 6, [{file, "d:/SPRTVL/source/sprtv1/_build/default/lib/reports/src/rp_rptpro.erl"}, {line, 1403}]}], [{lists, foldl, 3, [{file, "lists.erl"}, {line, 1263}]}], [{rp_rptpro, build_report, 1, [{file, "d:/SPRTVL/source/sprtv1/_build/default/lib/reports/src/rp_rptpro.erl"}, {line, 1621}]}], [{rp_rptpro, do_work, 1, [{file, "d:/SPRTVL/source/sprtv1/_build/default/lib/reports/src/rp_rptpro.erl"}, {line, 456}]}]}]
    ancestors: [rp_rptpro, reports, sprtv1, <0.15298.6305>]
    messages: []
    links: [<0.19948.6305>, <0.15259.6314>, <0.402.0>]
    dictionary: []
    trap_exit: true
    status: running
    heap_size: 4185
    stack_size: 27
    reductions: 10200429
  neighbours:
    neighbour:
      [{pid, <0.15259.6314>}, {registered_name, pdf}, {initial_call, {eg_pdf, init, ['Argument__1']}}, {current_function, {gen_server, loop, 6}}, {ancestors, [<0.8594.6314>, rp_rptpro, reports, sprtv1, <0.15298.6305>]}, {messages, []}, {links, [<0.8594.6314>]}, {dictionary, []}, {trap_exit, false}, {status, waiting}, {heap_size, 987}, {stack_size, 9}, {reductions, 18719}]].
```

- Next, we need to ensure that the dependencies are started in our `chp02` application, so we need to edit the `chp02_app.erl` and add the following two (2) lines to the `start(...)` function as show in Figure 54.

```
1  %%% -----
2  %%% @doc chp02 public API
3  %%% @end
4  %%% -----
5
6  -module(chp02_app).
7
8  -behaviour(application).
9
10 -export([start/2, stop/1]).
11
12 start(_StartType, _StartArgs) ->
13     application:start(sasl),
14     application:ensure_all_started(lager),
15     chp02_sup:start_link().
16
17 stop(_State) ->
18     ok.
19
20 %%% internal functions
21
```

Figure 54 - chp02_app add sasl, lager calls

8. Next, let's run our application by entering `./chp02.cmd console` on the command line in the `~/bootstrap/chp02` folder as show in Figure 55.

```

MINGW64:/c/Users/JSTARNES/bootcamp/chp02
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ ./chp02.cmd console

C:\Users\JSTARNES\bootcamp\chp02>._build/default/rel/chp02/bin/chp02.cmd console

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$

```

Figure 55 – ./chp02.cmd console

9. Next, let's review the log directory for the output files created by running the chp02 application in a console. Enter in the Git Bash shell the `'ls log'` command as shown in Figure 56.

```

MINGW64:/c/Users/JSTARNES/bootcamp/chp02
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ ls log
console.log crash.log debug.log error.log sasl/ sasl-error.log

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$

```

Figure 56 – ls log command

10. Now that we have logging, it is important to know the log levels support by Erlang and the methods for adding logging to our source code. The log levels are described in **Error! Reference source not found.**

Table 2 – Log Levels

Level	Integer	Description
Emergency	0	System is unusable
Alert	1	Action must be taken immediately
Critical	2	Critical conditions
Error	3	Error Conditions
Warning	4	Warning conditions
Notice	5	Normal but significant conditions
Info	6	Informational messages
Debug	7	Debug-level messages

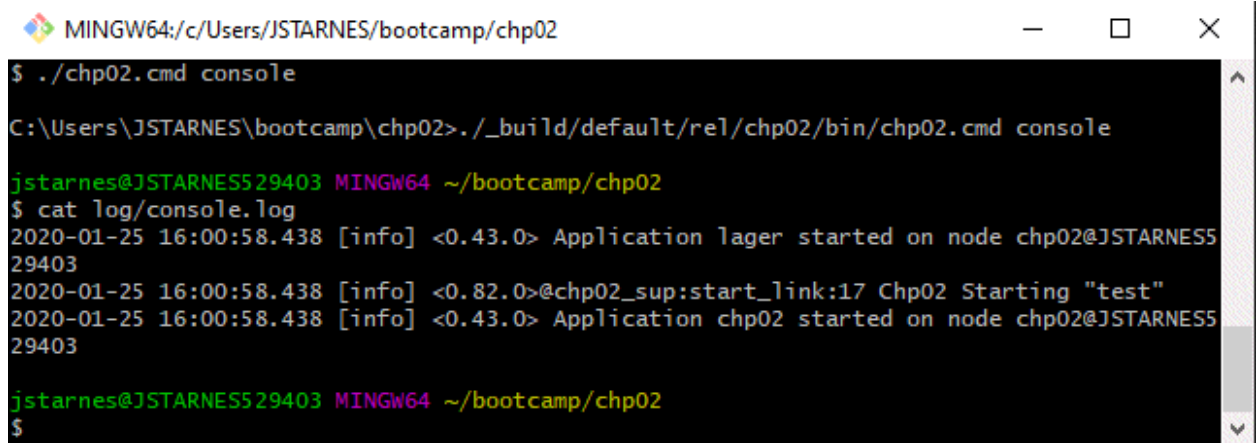
11. The lager library provides methods for logging these events using the lager functions and I suggest we use these for now. The Erlang OTP 21 logging has been restructured from the now deprecated `error_logger` functionality and has added structural logging, which shows much promise. However, for now we'll stick with lager and perhaps at latter chapter cover the new structural logging features of new logging capability.

12. Listing 12 shows an example usage of lager functions, note that support different arities dependent on need pass in arguments, the lager library magically adds the module:function:line# to the log which is very helpful.

Listing 12 – Lager Logging Function

```
lager:emergency("emergency message").
lager:alert("alert message").
lager:critical("notice message").
lager:error("error occurred reason: ~s", [Reason]).
lager:warning("warning message").
lager:notice("notice message").
lager:info("infor message").
lager:debug("debug message").
```

13. Next, let's at least add some informational log message in our chp02 application. Edit the chp02_sup.erl and add the "lager:info("Chp02 Starting ~p", ["test"]), to the first line underneath the start_link() -> function. Note, don't include the quotes as part of your edits in the chp02_sup.erl file. Run the rebar3 release to update source code.
14. Next, run the "./chp02.cmd console" on the command line in the "~/bootstrap/chp02" folder as shown in Figure 55. Now in your Git Bash shell, let's run the cat function on the contents of the "log/console.log" file as shown in Figure 57. Notice the Chp02 Starting "test" line, which shows the @chp02_sup:start_link:17.



```
MINGW64:/c:/Users/JSTARNES/bootcamp/chp02
$ ./chp02.cmd console
C:\Users\JSTARNES\bootcamp\chp02>._build/default/re1/chp02/bin/chp02.cmd console
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ cat log/console.log
2020-01-25 16:00:58.438 [info] <0.43.0> Application lager started on node chp02@JSTARNES529403
2020-01-25 16:00:58.438 [info] <0.82.0>@chp02_sup:start_link:17 Chp02 Starting "test"
2020-01-25 16:00:58.438 [info] <0.43.0> Application chp02 started on node chp02@JSTARNES529403
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$
```

Figure 57 – cat log/console.log command

15. One final word on logging, its important to not to log everything, too much detail tends to hide issues buried in logs. The lager library is best used for logging general application information and we should focus on capture warning and error conditons with very sparse if any informational detail. In a future chapter I will cover application specific logfiles that use an Erlang Term format that represents timestamp term data that can be reloaded for analysis purposes, etc.
16. Next, let's explore the SASL reports in the "./log/sasl" directory underneath the chp02 application folder. These log files are written by the error logger handler log_mf_h defined in STDLIB and this handler writes the multi-file reports to the directory defined using the error_logger_mf_dir parameter in the chp02 applications "config/sys.config" file in the SASL configuration section. If you were paying attention you may have noticed the new chp02_report.cmd command file in the output of the rebar3 new release chp02 in Figure 53. This file was added to the customized Erlang

Bootcamp release templates. Let's examine the contents of the file by using whatever editor you prefer as show in Figure 58.

```
1  @echo off
2  erl -smp -config config/sys.config -boot start_sasl -s rb
3  |
```

Figure 58 - Contents of chp02_report.cmd command file

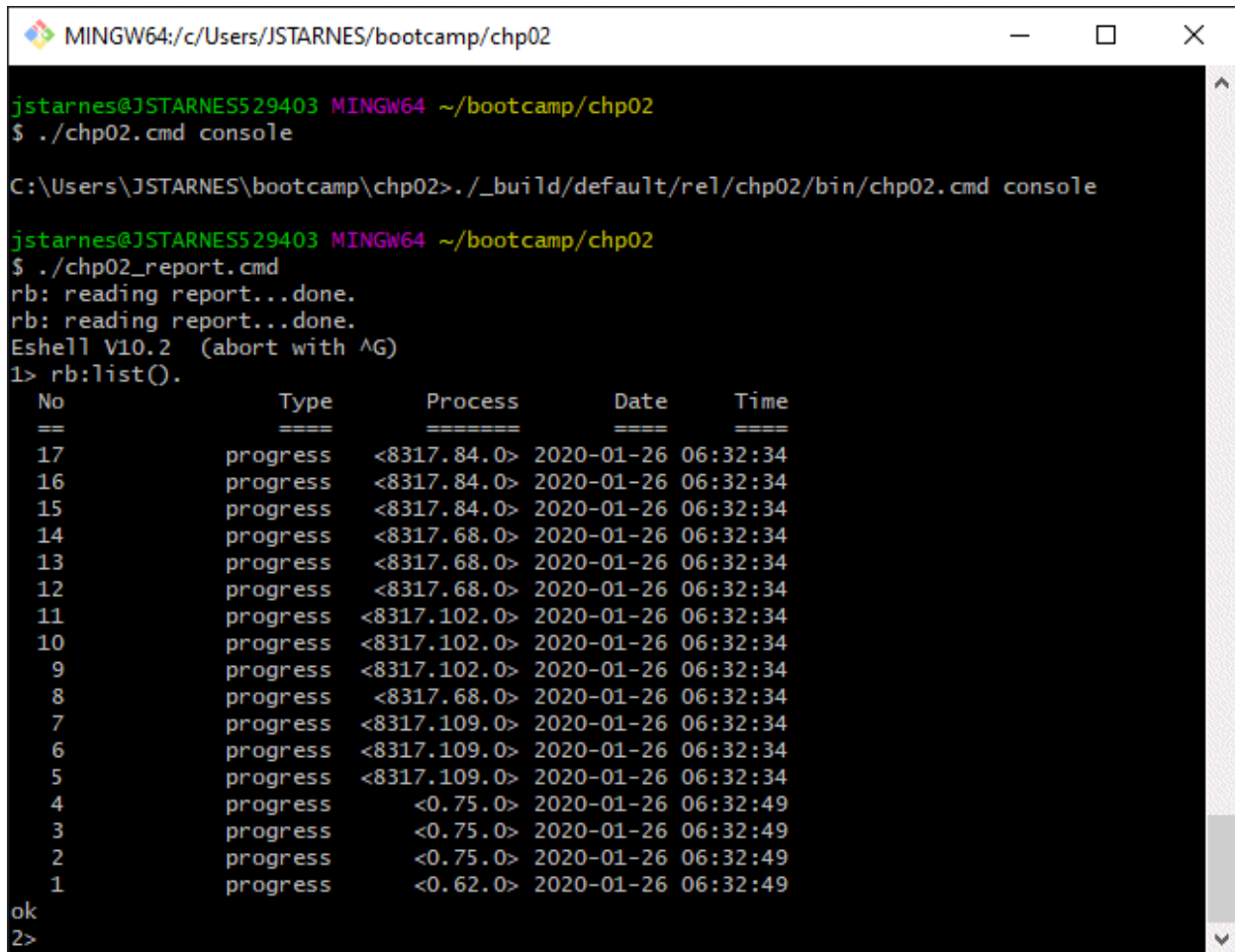
17. The chp02_report.cmd command file uses the same "config/sys.config" file as the chp02 application and launches a separate Erlang VM instances booting the SASL process and starting the rb application, aka the Report Browser. The Report Browser application is used to examine the contents of the log files generated by the error logger handler. If the chp02 application is running, let's exit it by first pressing CTRL+G key and then type q and return to quit. Now, let's run chp02_report.cmd as show in Figure 59.



```
MINGW64:/c/Users/JSTARNES/bootcamp/chp02
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ ./chp02_report.cmd
rb: reading report...done.
rb: reading report...done.
rb: reading report...done.
rb: reading report...Eshell V10.2 (abort with ^G)
1> done.
1>
```

Figure 59 – start ./chp02_report.cmd command file

18. The Report Browser loads the files underneath the "./log/sasl" directory and is ready to perform commands from the console. Let's get a list of the available messages by typing "rb:list()." and press return as show in Figure 60.



```

MINGW64:/c:/Users/JSTARNES/bootcamp/chp02
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ ./chp02.cmd console

C:\Users\JSTARNES\bootcamp\chp02>./_build/default/rel/chp02/bin/chp02.cmd console

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ ./chp02_report.cmd
rb: reading report...done.
rb: reading report...done.
Eshell V10.2 (abort with ^G)
1> rb:list().

```

No	Type	Process	Date	Time
17	progress	<8317.84.0>	2020-01-26	06:32:34
16	progress	<8317.84.0>	2020-01-26	06:32:34
15	progress	<8317.84.0>	2020-01-26	06:32:34
14	progress	<8317.68.0>	2020-01-26	06:32:34
13	progress	<8317.68.0>	2020-01-26	06:32:34
12	progress	<8317.68.0>	2020-01-26	06:32:34
11	progress	<8317.102.0>	2020-01-26	06:32:34
10	progress	<8317.102.0>	2020-01-26	06:32:34
9	progress	<8317.102.0>	2020-01-26	06:32:34
8	progress	<8317.68.0>	2020-01-26	06:32:34
7	progress	<8317.109.0>	2020-01-26	06:32:34
6	progress	<8317.109.0>	2020-01-26	06:32:34
5	progress	<8317.109.0>	2020-01-26	06:32:34
4	progress	<0.75.0>	2020-01-26	06:32:49
3	progress	<0.75.0>	2020-01-26	06:32:49
2	progress	<0.75.0>	2020-01-26	06:32:49
1	progress	<0.62.0>	2020-01-26	06:32:49

```

ok
2>

```

Figure 60 - chp02 rb:list(). command

19. As you can see there are seventeen (17) progress reports in the log. Let's examine the first, tenth, and seventeenth items by issuing the following Report Browser commands in succession, "rb:show(1). rb:show(10). rb:show(17)." as seen in Figure 61.

```

MINGW64:/c:/Users/JSTARNES/bootcamp/chp02
2> rb:show(1).

PROGRESS REPORT    <0.43.0>                                2020-01-26 06:32:49
=====
application                                sasl
started_at                                nonode@nohost

ok
3> rb:show(10).

PROGRESS REPORT    <8317.104.0>                             2020-01-26 06:32:34
=====
supervisor                                {local,gr_sup}
started
    [{pid,<8317.106.0>},
     {id,gr_param_sup},
     {mfargs,{gr_param_sup,start_link,[]}},
     {restart_type,permanent},
     {shutdown,5000},
     {child_type,supervisor}]

ok
4> rb:show(17).

PROGRESS REPORT    <8317.95.0>                             2020-01-26 06:32:34
=====
supervisor                                {local,sasl_safe_sup}
started
    [{pid,<8317.96.0>},
     {id,alarm_handler},
     {mfargs,{alarm_handler,start_link,[]}},
     {restart_type,permanent},
     {shutdown,2000},
     {child_type,worker}]

ok
5>

```

Figure 61 - chp02 `rb:show(1)`, `rb:show(10)` and `rb:show(17)` commands

20. The SASL application provides the following three (3) types of reports:

- **Supervisor Report** – A supervisor report is issued when a supervised child terminates in an unexpected way. A supervisor report contains the following items:
 - **Supervisor** – The name of the reporting supervisor.
 - **Context** – Indicates the phase the child terminated from the supervisor's point of view. This can be `start_error`, `child_terminated`, or `shutdown_error`.
 - **Reason** – The termination reason.
 - **Offender** – The start specification of the child.
- **Progress Report** – A progress report is issued whenever a supervisor starts or restarts. A progress report contains the following items:
 - **Supervisor** – The name of the reporting supervisor.
 - **Started** – The start specification for the successfully started child.
- **Crash Report** – Processed started with the `proc_lib:spawn` or `proc_lib:spawn_link` functions are wrapped with a `catch`. A crash report is issued whenever such a process

terminates with an unexpected reason, which is any reason other than normal or shutdown. Processes using `get_server` and `gen_fsm/gen_statem` behaviors are examples of such a process. A crash report contains the following items:

- **Crasher** – Information about the crashing process is reported, such as initial function call, exit reason, and message queue.
 - **Neighbors** – Information about the processes which are linked to the crashing process and do not trap exits. These processes are the neighbors which will terminate because of this process crash. The information gathered is the same as the information for Crasher, shown in previous item.
21. The SASL application reports can be confusing given that we added the lager library which intercepts some of the Crash Reports to produce the “./log/crash.log” output. The lager library is used for the convenience of reporting this content is a textual log with a rotation. However, there will be instances whereas the SASL application reports would be your best choice to understand a startup failure by reviewing the Progress Reports.
22. Logging plays a major role in the Erlang in Production based on the presentation by Jesper Loius Andersen and Erlang SASL application reports add the critical system information for the logs. There is an excellent blog post “The Extinction of the Dodos (OTP Style)” [5] by Brujo Benavides that illustrates usage of the SASL application reports to understand why `world:dodos` went extinct in his example Erlang OTP application.
23. Next topic for the Erlang in Production is alarm handling. Alarms often get confused with events which just state that something happened, an alarm states that something is wrong with the system. Once an alarm is raised it remains until the error condition no longer applies. Typically when the error condition is removed the associated alarm is cleared. That being said, alarm logs themselves represent the history of the creation, any state changes and resolution of an alarm instances and they're technically events. It can be confusing at times; however, by following a general guideline of only making alarms that represent an error state for your application.
24. We are going to use the `elarm` library as our alarming solution as it adds a number of key features that every alarm system needs. The following is a list of key features:
- Alarm List – Is a list of all the currently active alarms and it is possible to subscribe to all changes of the alarms.
 - Alarm Log – All changes to an alarm are logged in an alarm log.
 - Alarm Configuration - Alarm configuration can be optionally added via `elarm:add_configuration/2`.
 - Application raises an alarm using `elarm:raise` and clears an alarm with `elarm:clear`
 - Subscriptions are supported with filtering using the `elarm:subscriber(Server, Filter)` call.
 - Alarm Summary can be subscribed for using the `elarm:subscribe_summary(Server, Filter)` call.
 - Acknowledge Alarms using the `elarm:acknowledge/3` call and is timestamp with `UserId`
 - Comment Alarms comments can be added to alarms
 - Clear Alarms
 - `elarm:clear` – Automatic clear of alarm.
 - `elarm:manual_clear` – Manual clear

25. Next, let's add the alarm repo to our chp02 application in the rebar.config file and add the alarm module to the list of modules as shown in Listing 13.

Listing 13 - add alarm repo to rebar.config

```
{erl_opts, [debug_info, {parse_transform, lager_transform}]}].
{deps, [
  %% Packages
  {lager, "3.6.3"},
  %% Source Dependencies
  {alarm, {git, "https://github.com/esl/alarm.git", {branch, "master"}}}
]}.

{relx, [{release, {chp02, "0.1.0"},
  [chp02, sasl, lager, alarm]},

  {sys_config, "./config/sys.config"},
  {vm_args, "./config/vm.args"},

  {dev_mode, false},
  {include_erts, false},
  {system_libs, false},
  {extended_start_script, true},
  {overlay, [{mkdir, "./log/sasl"}]}]}.

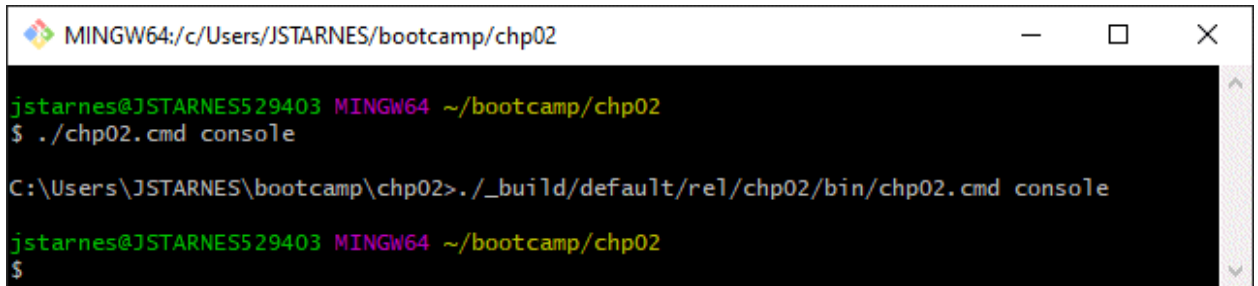
{profiles, [{prod, [{relx, [{dev_mode, false},
  {include_erts, true},
  {system_libs, true},
  {include_src, false},
  {debug_info, strip}]}]}]}.
}
```

26. Next, we need to add the alarm module on chp02 application startup, so let's edit the chp02_app.erl file as shown in Figure 62.

```
1  %%%-----
2  %%% @doc chp02 public API
3  %%% @end
4  %%%-----
5
6  -module(chp02_app).
7
8  -behaviour(application).
9
10 -export([start/2, stop/1]).
11
12 start(_StartType, _StartArgs) ->
13     application:start(sasl),
14     application:ensure_all_started(lager),
15     application:ensure_all_started(alarm),
16     chp02_sup:start_link().
17
18 stop(_State) ->
19     ok.
20
21 %%% internal functions
22
```

Figure 62 - Add elarm module to startup of chp02 application

27. Next, let's run our application by entering `./chp02.cmd console` on the command line in the `~/bootstrap/chp02` folder as show in Figure 63.



```

MINGW64:/c/Users/JSTARNES/bootcamp/chp02
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ ./chp02.cmd console

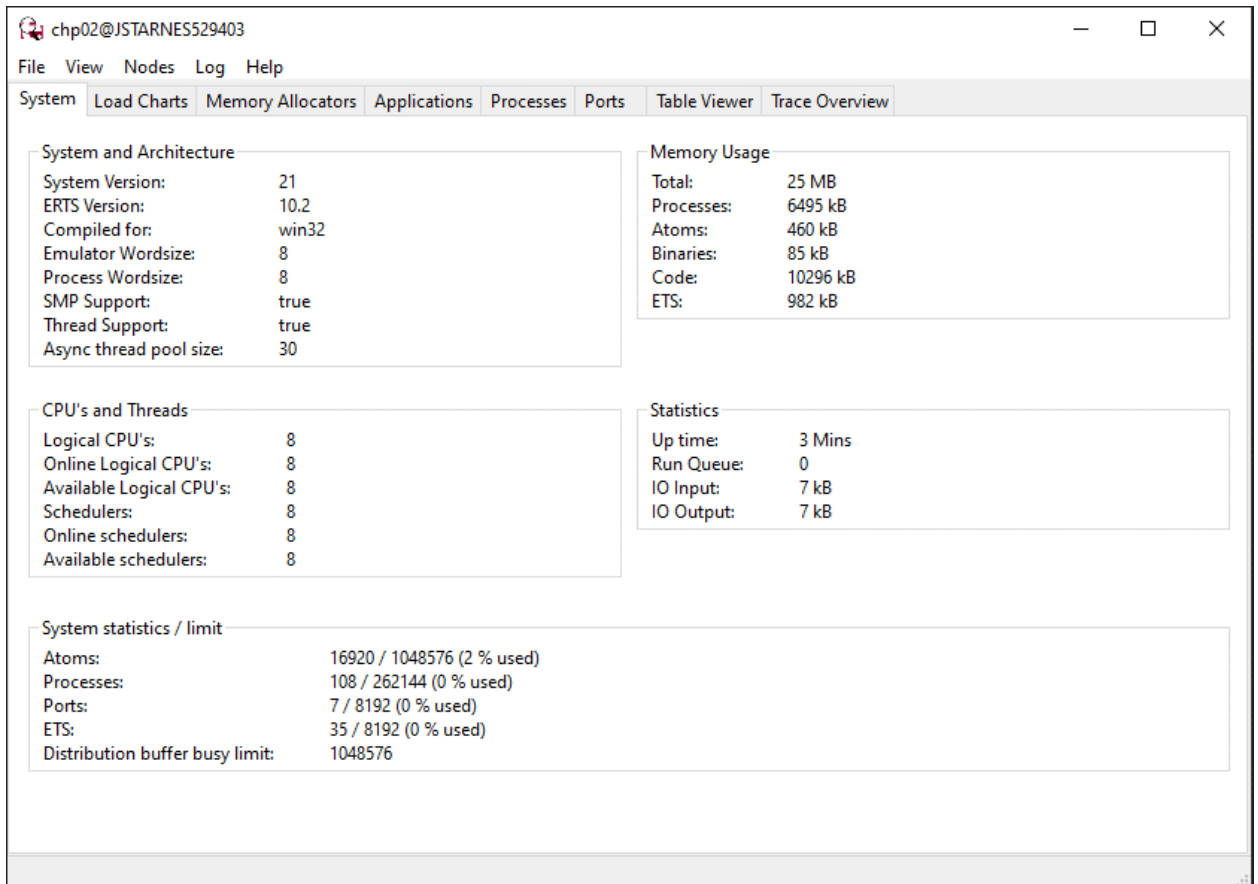
C:\Users\JSTARNES\bootcamp\chp02>._build/default/rel/chp02/bin/chp02.cmd console

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$

```

Figure 63 - ./ch02 cmd console

28. Next, move your cursor in the console window and enter the following command `observer:start()`. at the `(chp02@<Nodename>)1>` prompt. The observer application starts and the System tab is active as show in Figure 64. The Observer application provides an interface for monitoring of active Erlang OTP application and is one of the tools used in development and production to gain knowledge about the operations of the Erlang application at runtime.

**Figure 64 - chp02 Observer System tab**

29. Next, let's first click the Applications tab, the Applications screen is divided into two panes, the first pane is the list of top level applications and the second pane is a graphical representation of the process tree of the selected application in the first pane. Next, click on each top level application in the left pane and see the changes in the right pane process tree graph. Note that the chp02 is supported by the other top level applications, such as the lager and elarm modules. Also, note that other applications such as goldrush are dependencies brought into chp02 indirectly by inclusion of of an other application that depends on it, in this case the lager library. In future chapters we will discuss in greater detail the composition of an Erlang OTP application and when it is appropriate to create top-level applications vs embedded applications that are part of the process tree of your main application. Figure 65 shows the Elarm top level application graphical process tree. The processes in the graphical process tree are either named or have an associated Process ID (PID) in the form of <A.B.C> [whereas](#):

- A, the node number (0 is the local node, an arbitrary number for a remote node)
- B, the first 15 bits of the process number (an index into the process table)
- C, bits 16-18 of the process number (the same process number as B)

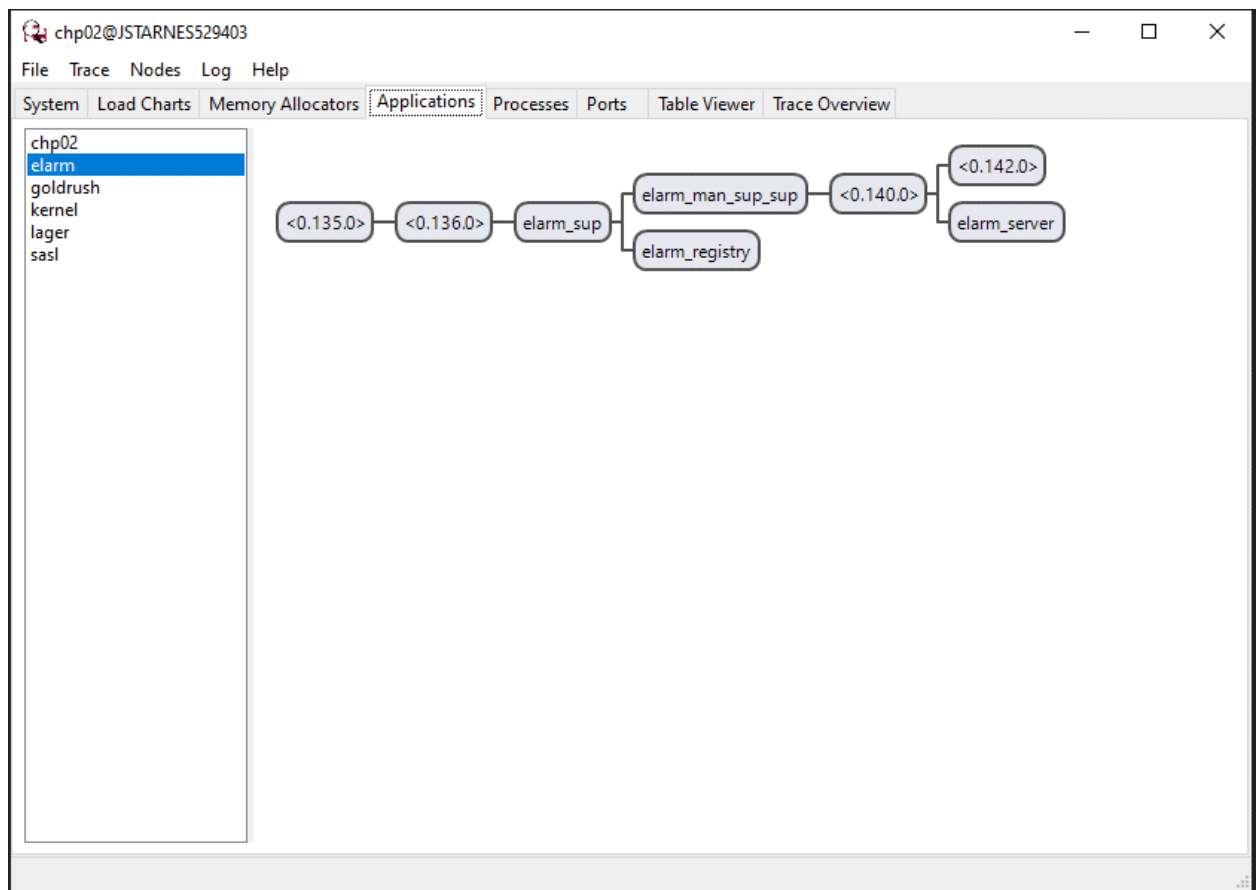
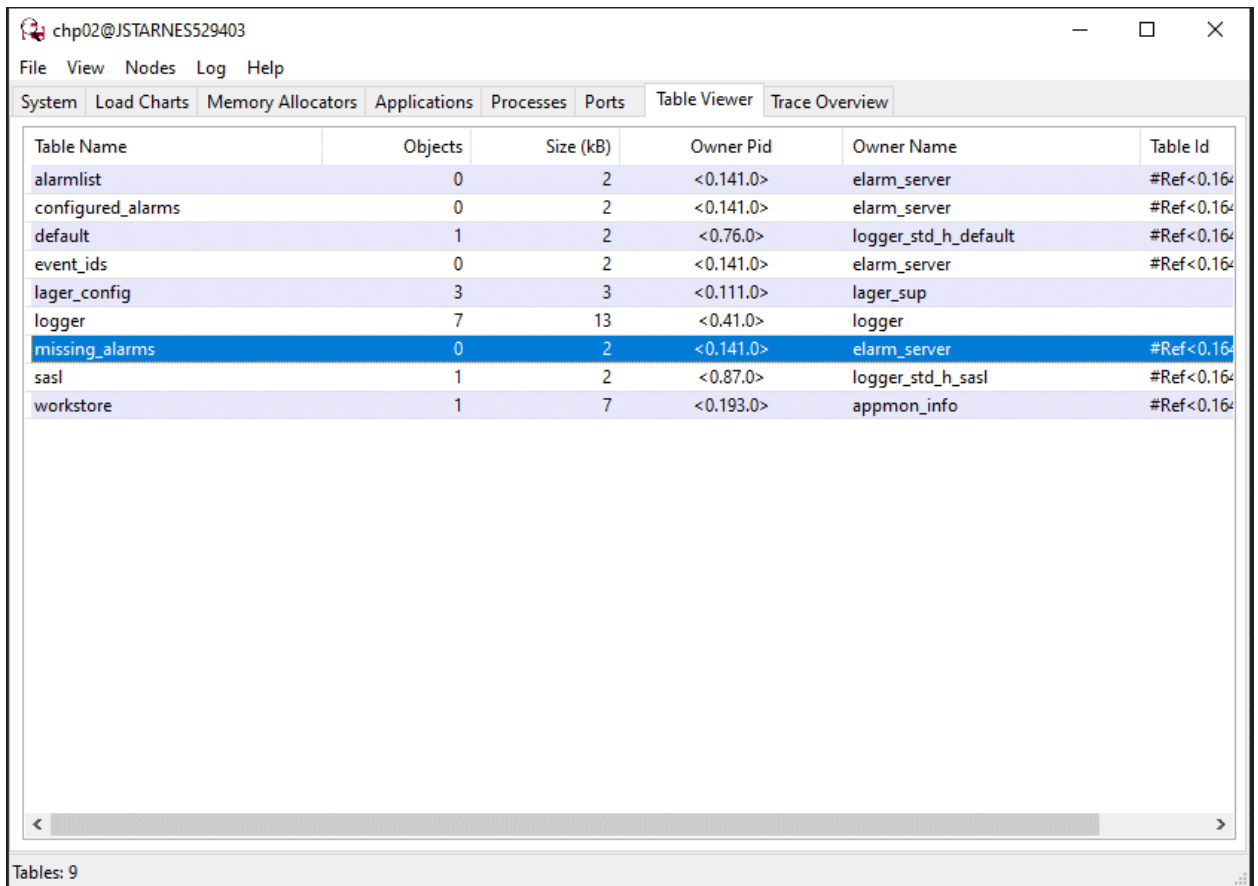


Figure 65 - chp2 Observer Applications Elarm

30. Next, click on the Table Viewer tab, the Table Viewer supports both Erlang Term Storage (ETS) and Mnesia tables. The Elarm application creates the following tables: *alarmlist*, *configured_alarms*, *event_ids*, and *missing_alarms* as shown in Figure 66.



chp02@JSTARNES529403

File View Nodes Log Help

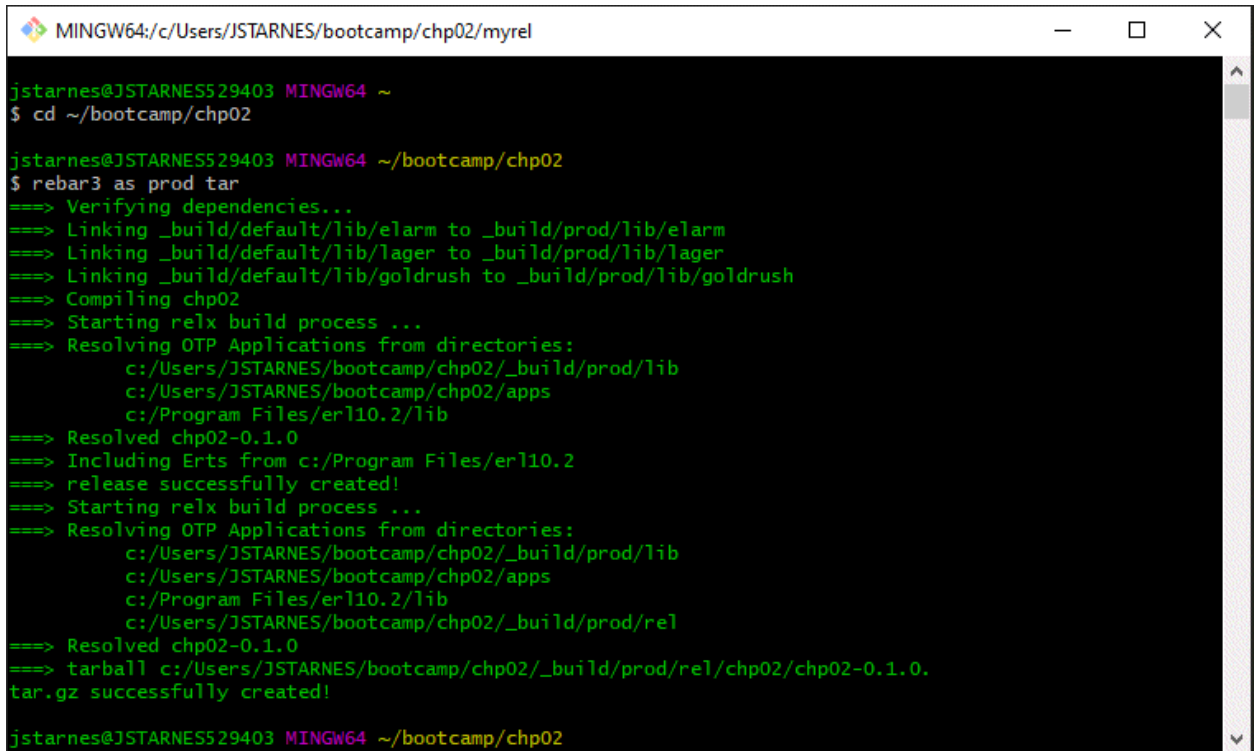
System Load Charts Memory Allocators Applications Processes Ports Table Viewer Trace Overview

Table Name	Objects	Size (kB)	Owner Pid	Owner Name	Table Id
alarmlist	0	2	<0.141.0>	alarm_server	#Ref<0.164...
configured_alarms	0	2	<0.141.0>	alarm_server	#Ref<0.164...
default	1	2	<0.76.0>	logger_std_h_default	#Ref<0.164...
event_ids	0	2	<0.141.0>	alarm_server	#Ref<0.164...
lager_config	3	3	<0.111.0>	lager_sup	
logger	7	13	<0.41.0>	logger	
missing_alarms	0	2	<0.141.0>	alarm_server	#Ref<0.164...
sasl	1	2	<0.87.0>	logger_std_h_sasl	#Ref<0.164...
workstore	1	7	<0.193.0>	appmon_info	#Ref<0.164...

Tables: 9

Figure 66 - chp2 Observer Table Viewer Alarm

31. It's a good idea to periodically during development to build a prod release to insure that some critical configuration is not missing when adding features to your application. Let's do that now, but first close the observer application and in the console window press CTRL+G followed by q character and press enter. Now, let's build the prod release by entering the rebar3 as prod tar command as shown in Figure 67.



```

MINGW64:/c/Users/JSTARNES/bootcamp/chp02/myrel
jstarnes@JSTARNES529403 MINGW64 ~
$ cd ~/bootcamp/chp02

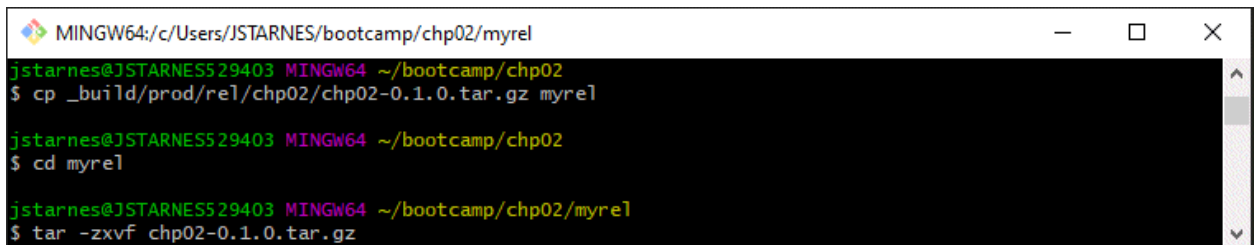
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ rebar3 as prod tar
==> Verifying dependencies...
==> Linking _build/default/lib/etarm to _build/prod/lib/etarm
==> Linking _build/default/lib/lager to _build/prod/lib/lager
==> Linking _build/default/lib/goldrush to _build/prod/lib/goldrush
==> Compiling chp02
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
      c:/Users/JSTARNES/bootcamp/chp02/_build/prod/lib
      c:/Users/JSTARNES/bootcamp/chp02/apps
      c:/Program Files/erl10.2/lib
==> Resolved chp02-0.1.0
==> Including Ert from c:/Program Files/erl10.2
==> release successfully created!
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
      c:/Users/JSTARNES/bootcamp/chp02/_build/prod/lib
      c:/Users/JSTARNES/bootcamp/chp02/apps
      c:/Program Files/erl10.2/lib
      c:/Users/JSTARNES/bootcamp/chp02/_build/prod/re1
==> Resolved chp02-0.1.0
==> tarball c:/Users/JSTARNES/bootcamp/chp02/_build/prod/re1/chp02/chp02-0.1.0.
tar.gz successfully created!

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02

```

Figure 67 - chp02 rebar3 as prod tar

32. Next, copy the chp02-0.1.0.tar.gz to your myrel directory as shown in Figure 68.



```

MINGW64:/c/Users/JSTARNES/bootcamp/chp02/myrel
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ cp _build/prod/re1/chp02/chp02-0.1.0.tar.gz myrel

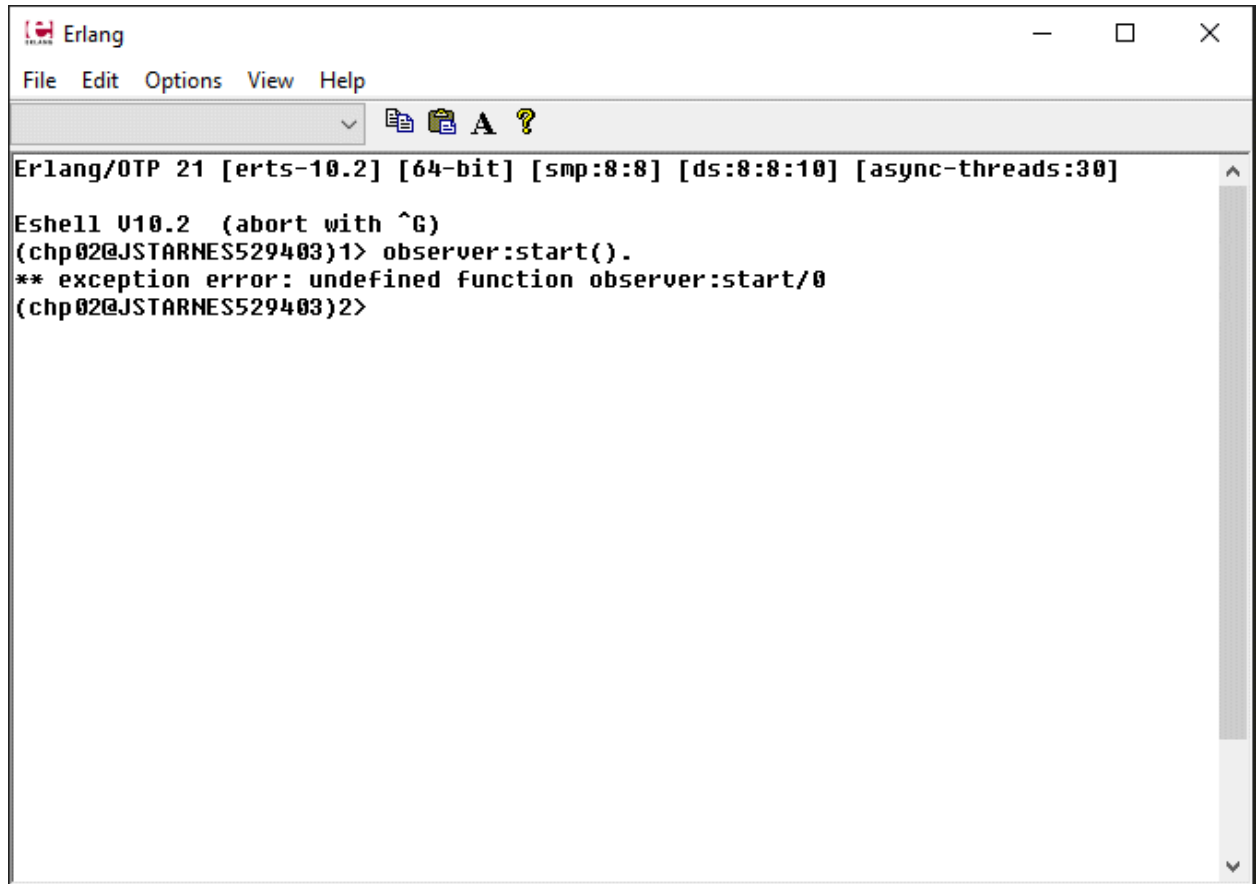
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ cd myrel

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02/myrel
$ tar -zxvf chp02-0.1.0.tar.gz

```

Figure 68 - chp02 copy chp02-0.1.0.tar.gz to myrel

33. Next, let's run the bin/chp02.cmd console command and start the observer application as shown in Figure 69.



```

Erlang
File Edit Options View Help
Erlang/OTP 21 [erts-10.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:30]
Eshell V10.2 (abort with ^G)
(chp02@JSTARNES529403)1> observer:start().
** exception error: undefined function observer:start/0
(chp02@JSTARNES529403)2>

```

Figure 69 - chp02 prod console failed to run observer

Oops, what happened here? Actually it is really simple; the prod releases only contain components you include in the rebar.config, so we are missing observer and its dependencies. Let's fix that issue by editing the chp02 rebar3.config and add the three (3) of the missing dependencies: *runtime_tools*, *wx*, and *observer* as shown in Listing 14.

Listing 14, *Note the rebar.config kernel and stdlib items are mandatory for Erlang applications, so we add them for clarity.*

Listing 14 - chp02 prod rebar.config updated for observer application

```

{erl_opts, [debug_info, {parse_transform, lager_transform}]}.
{deps, [
  %% Packages
  {lager, "3.6.3"},
  %% Source Dependencies
  {elarm, {git, "https://github.com/esl/elarm.git", {branch, "master"}}}
]}.

{relx, [{release, {chp02, "0.1.0"},
  [kernel, stdlib, sasl, runtime_tools,
   wx, observer, chp02, lager, elarm]},

  {sys_config, "./config/sys.config"},
  {vm_args, "./config/vm.args"},

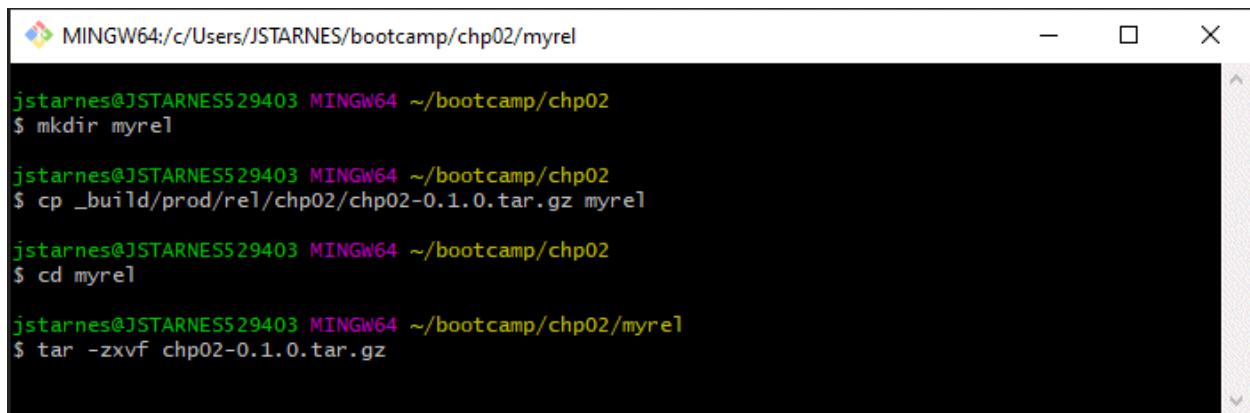
  {dev_mode, false},
  {include_erts, false},
  {system_libs, false},
  {extended_start_script, true},

```

```
        {overlay, [{mkdir, "./log/sasl"}]}}
    }.

    {profiles, [{prod, [{relx, [{dev_mode, false},
                                {include_erts, true},
                                {system_libs, true},
                                {include_src, false},
                                {debug_info, strip}]}]}]}
    }.
}
```

34. Next, let's create a myrel folder underneath chp02 folder to install our production release and copy the generated chp02-0.1.0.tar.gz into that directory and then run the tar -zxvf command to extract the release archive as shown in Figure 70.



```
MINGW64:/c:/Users/JSTARNES/bootcamp/chp02/myrel
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ mkdir myrel

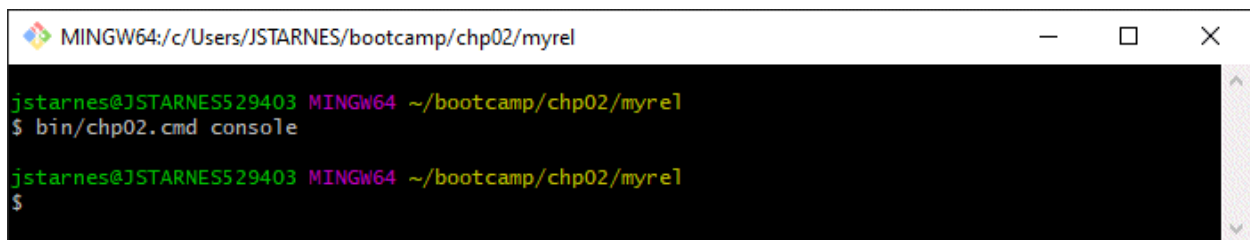
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ cp _build/prod/rel/chp02/chp02-0.1.0.tar.gz myrel

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ cd myrel

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02/myrel
$ tar -zxvf chp02-0.1.0.tar.gz
```

Figure 70 - chp02 install prod release

35. Next, let's run the prod release of chp02 application by entering "bin/chp02.cmd console" on the command line in the "~/bootstrap/chp02myrel" folder as show in Figure 71.



```
MINGW64:/c:/Users/JSTARNES/bootcamp/chp02/myrel
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02/myrel
$ bin/chp02.cmd console

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02/myrel
$
```

Figure 71 - bin/chp02.cmd console

36. Next, let's start the observer application in the console and verify it now works as shown in Figure 72.

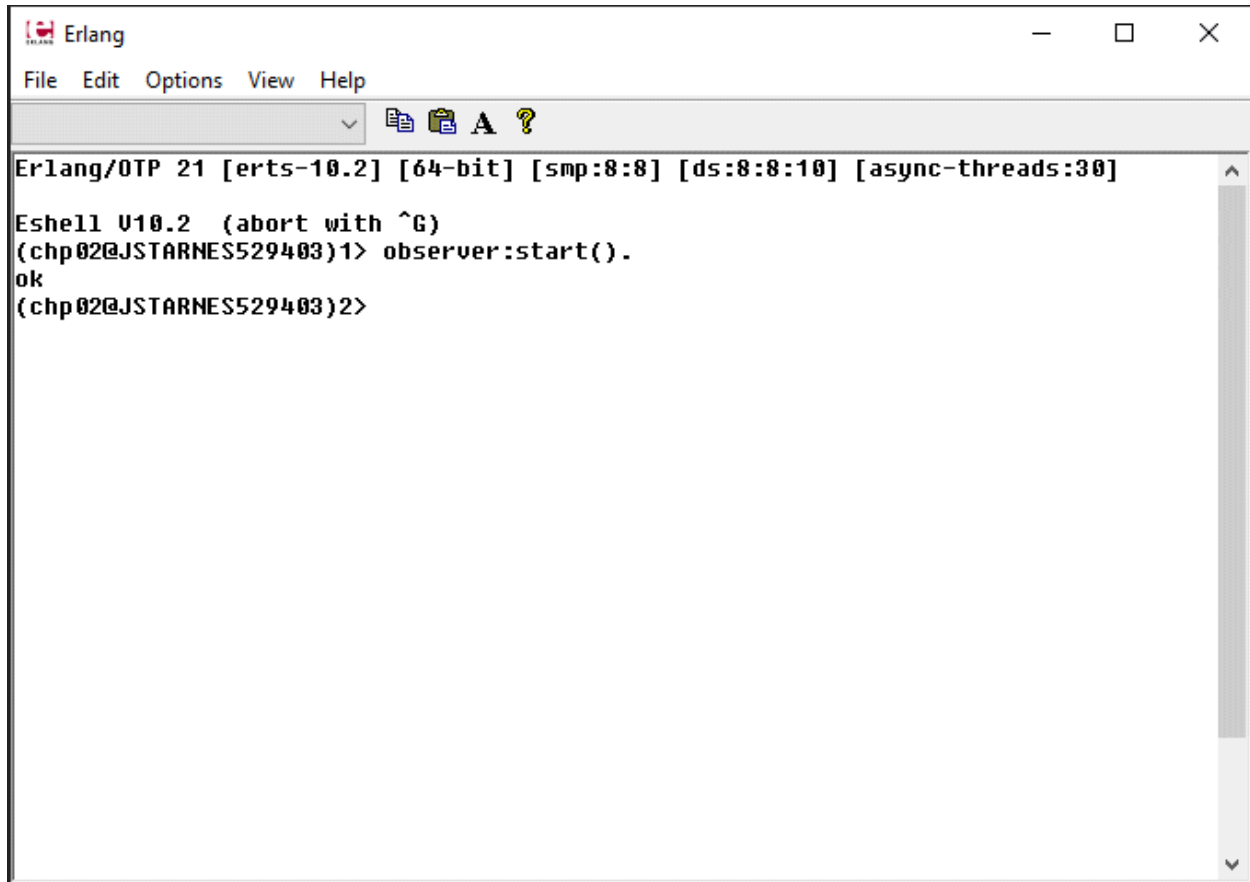


Figure 72 - chp02 prod release console observer

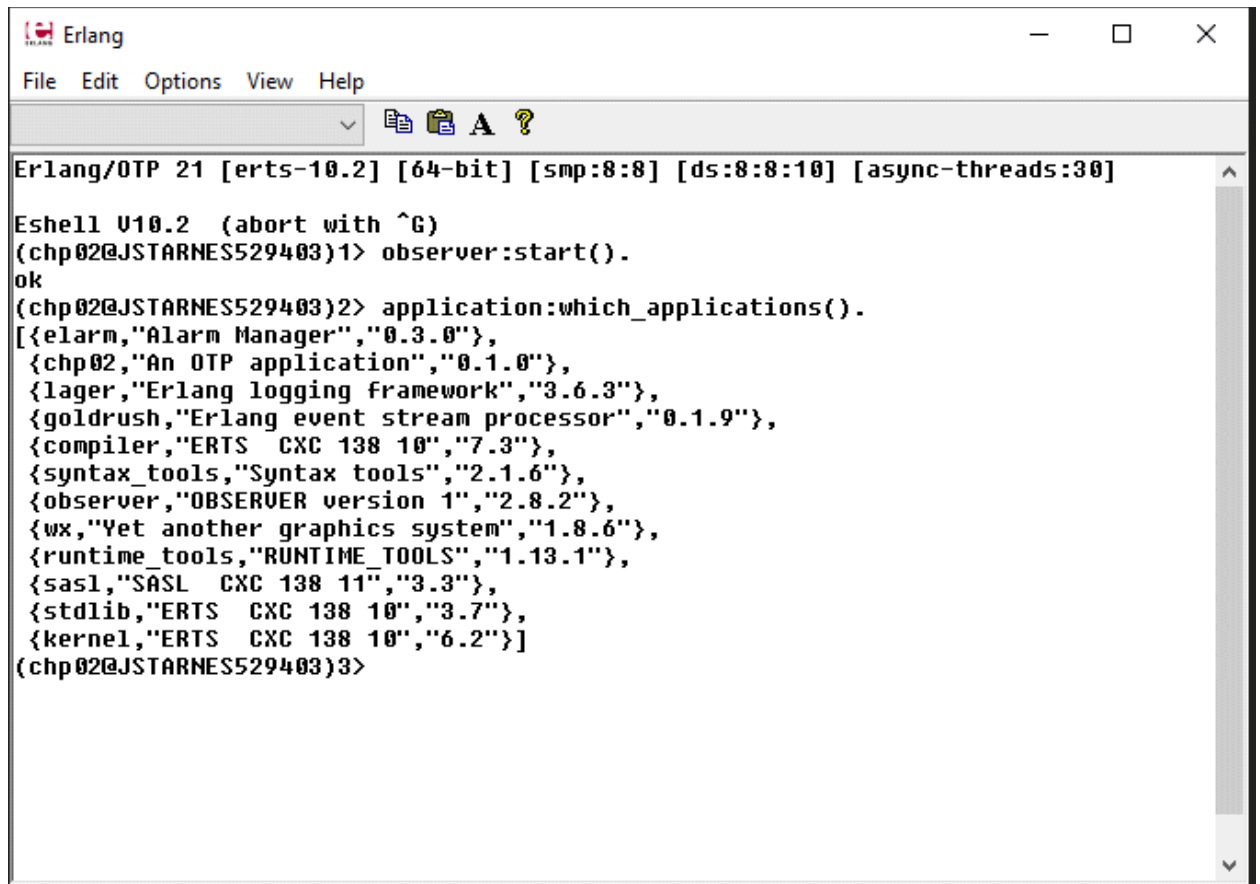
37. Next, let's take another look at the observer and click on the Processes tab. The Processes tab shows a table of all the processes running in the selected Node. The table has the following columns: *Pid*, *Name or Initial Func*, *Reds*, *Memory*, *MsgQ*, and *Current Function*. The Processes bares some similarity to Windows Task Manager Processes, both have PID (Process ID) and Memory. The Processes Reds column represents the number of reductions performed on a process. Reductions equate roughly to function calls and the scheduler in Erlang allows for two thousand (2000) reductions before switching to another process. This is how Erlang can run thousands or even millions of processes concurrently. The processes are re-entrant and distributed across all processor cores. The size of a process in Erlang equates to about the same memory usage a C++ object. The MsgQ column represents the number of queued messages in the processes mailbox. This is where your need to check if messages are accumulating due to some issue, ideally this value stays close or at 0 most of the time. Figure 73 shows the Processes table in the Observer application.

Pid	Name or Initial Func	Reds	Memory	MsgQ	Current Function
<0.1077.0>	observer_backend:fetch_stats/2	36923	210928	0	observer_backend:fetch_stats_loop/2
<0.230.0>	wxe_server:init/1	33641	110336	0	gen_server:loop/7
<0.238.0>	observer_alloc_wx:init/1	9817	1804104	0	wx_object:loop/6
<0.241.0>	erlang:apply/2	9590	196848	0	observer_pro_wx:table_holder/1
<0.237.0>	observer_perf_wx:init/1	4395	284668	0	wx_object:loop/6
<0.232.0>	erlang:apply/2	2495	24476	0	timer:sleep/1
<0.71.0>	kernel_refc	429	5800	0	gen_server:loop/7
<0.240.0>	observer_pro_wx:init/1	375	11920	0	wx_object:loop/6
<0.87.0>	timer_server	136	24820	0	gen_server:loop/7
<0.84.0>	logger_std_h_sasl	93	27916	0	gen_server:loop/7
<0.229.0>	observer	70	122440	0	wx_object:loop/6
<0.58.0>	net_kernel	16	8968	0	gen_server:loop/7
<0.60.0>	net_kernel:ticker/2	7	2644	0	net_kernel:ticker_loop/2
<0.45.0>	application_master:init/4	2	3920	0	application_master:main_loop/2
<0.66.0>	supervisor_bridge:user_sup/1	2	8808	0	gen_server:loop/7
<0.70.0>	kernel_config:init/1	2	2732	0	gen_server:loop/7
<0.81.0>	application_master:init/4	2	2776	0	application_master:main_loop/2
<0.86.0>	raw_file_io_delayed:init/1	2	7052	0	gen_statem:loop_receive/3
<0.97.0>	application_master:init/4	2	2776	0	application_master:main_loop/2
<0.104.0>	application_master:init/4	2	2776	0	application_master:main_loop/2
<0.109.0>	application_master:init/4	2	2776	0	application_master:main_loop/2
<0.116.0>	application_master:init/4	2	2776	0	application_master:main_loop/2
<0.122.0>	lager_handler_watcher:init/1	2	2776	0	gen_server:loop/7
<0.124.0>	lager_handler_watcher:init/1	2	3920	0	gen_server:loop/7

Number of Processes: 110

Figure 73 - Observer Processes Table

38. Next, let's close the Observer application and return the to Console window and type the following command "application:which_applications()." as shown in Figure 74. The application:which_applications(). Function call lists the active applications running in the chp02 application. It should be clear to you now that Erlang applications are composed of multiple applications. Moreover, in a console window or remote shell you can execute public functions of any application and or library that is part of the chp02 application. The Erlang VM allows for the interspection of the Erlang VM and provides tools that aid in management of a running system.



```
Erlang/OTP 21 [erts-10.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:30]

Eshell V10.2 (abort with ^G)
(chp02@JSTARNES529403)1> observer:start().
ok
(chp02@JSTARNES529403)2> application:which_applications().
[{elarm,"Alarm Manager","0.3.0"},
 {chp02,"An OTP application","0.1.0"},
 {lager,"Erlang logging framework","3.6.3"},
 {goldrush,"Erlang event stream processor","0.1.9"},
 {compiler,"ERTS CXC 138 10","7.3"},
 {syntax_tools,"Syntax tools","2.1.6"},
 {observer,"OBSERVER version 1","2.8.2"},
 {wx,"Yet another graphics system","1.8.6"},
 {runtime_tools,"RUNTIME_TOOLS","1.13.1"},
 {sasl,"SASL CXC 138 11","3.3"},
 {stdlib,"ERTS CXC 138 10","3.7"},
 {kernel,"ERTS CXC 138 10","6.2"}]
(chp02@JSTARNES529403)3>
```

Figure 74 - chp02 console application:which_applications().

39. Now that we got the Elarm library in our chp02 application, let's issue our first alarm by entering the following "elarm:raise(test, "Src", [])."
- in the console and restart the observer application window as show in Figure 75.

```

Erlang
File Edit Options View Help

Erlang/OTP 21 [erts-10.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:30]

Eshell V10.2 (abort with ^G)
(chp02@JSTARNES529403)1> observer:start().
ok
(chp02@JSTARNES529403)2> application:which_applications().
[{elarm,"Alarm Manager","0.3.0"},
 {chp02,"An OTP application","0.1.0"},
 {lager,"Erlang logging framework","3.6.3"},
 {goldrush,"Erlang event stream processor","0.1.9"},
 {compiler,"ERTS CXC 138 10","7.3"},
 {syntax_tools,"Syntax tools","2.1.6"},
 {observer,"OBSERVER version 1","2.8.2"},
 {wx,"Yet another graphics system","1.8.6"},
 {runtime_tools,"RUNTIME_TOOLS","1.13.1"},
 {sas1,"SASL CXC 138 11","3.3"},
 {stdlib,"ERTS CXC 138 10","3.7"},
 {kernel,"ERTS CXC 138 10","6.2"}]
(chp02@JSTARNES529403)3> elarm:raise(test, "Src", []).
ok
(chp02@JSTARNES529403)4> observer:start().
ok
(chp02@JSTARNES529403)5> █

```

Figure 75 - chp02 console raise:alarm().

40. Next, go to the Observer application and click the Table Viewer and examine the following tables: *alarmlist*, *configured_alarms*, *event_ids*, and *missing_alarms*. Notice that the tables have objects in them except the *configured_alarms*. Now select the *alarmlist* and right click and select Show Table Content from the popip menu as shown in Figure 76.

1	2
{test,"Src"}	{alarm,test,undefined,"Src",{{2020,1,29},{1,46,9}},{{1580,262369,384000},-576460752303423481},indeterminate,<"..."

Objects: 1

Figure 76 - chp02 Table Viewer ets:alarmlist table

41. The contents of the *alarmlist* is divided into table columns, the first represents the unique key for the alarm and the second column represents the alarm records contents. If we were to raise

another alarm with the same key, then the existing record timestamp would be updated. Note that a lot of the fields in the alarm record are set to either indeterminate, undefined, [], and <<>>. Some of these are due to the fact that this alarm is not configured in elarms tables. That information is in the configured_elarms table. The elarm application allows for the generation of alarms regardless if the configuration exists. Moreover, when new unconfigured alarms are introduced the elarm application keeps track of the new alarms in the missing_elarms table. Let's repeat the last raise alarm command by going into the console window and press the up arrow twice and backspace out the "[]," so we can add a [2]). as the last argument as shown in Figure 77.

```

Erlang
File Edit Options View Help
Erlang/OTP 21 [erts-10.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:30]
Eshell V10.2 (abort with ^G)
(chp02@JSTARNES529403)1> observer:start().
ok
(chp02@JSTARNES529403)2> application:which_applications().
[{elarm,"Alarm Manager","0.3.0"},
 {chp02,"An OTP application","0.1.0"},
 {lager,"Erlang logging framework","3.6.3"},
 {goldrush,"Erlang event stream processor","0.1.9"},
 {compiler,"ERTS CXC 138 10","7.3"},
 {syntax_tools,"Syntax tools","2.1.6"},
 {observer,"OBSERVER version 1","2.8.2"},
 {wx,"Yet another graphics system","1.8.6"},
 {runtime_tools,"RUNTIME_TOOLS","1.13.1"},
 {sas1,"SASL CXC 138 11","3.3"},
 {stdlib,"ERTS CXC 138 10","3.7"},
 {kernel,"ERTS CXC 138 10","6.2"}]
(chp02@JSTARNES529403)3> elarm:raise(test, "Src", []).
ok
(chp02@JSTARNES529403)4> observer:start().
ok
(chp02@JSTARNES529403)5> elarm:raise(test, "Src", [2]).
ok
(chp02@JSTARNES529403)6>

```

Figure 77 - chp02 elarm:raise(test, "Src", [2]).

42. Next, go back to the TV ets:alarmlist view and go to the Edit menu and select Refresh menu item and then double click the first row to bring up the Edit object dialog as shown in Figure 78. Notice the value of [2] is now in the alarm record. This is an important feature of the Elarm application as this represents structural logging, where relevant data can be added to the alarm at the source of the alarm and later retrieved to generate very detailed alarm information. In a future chapter we will expand on this feature and add django templates via the erlydtl library that will use the additional information and build meaningful alarms with ease.

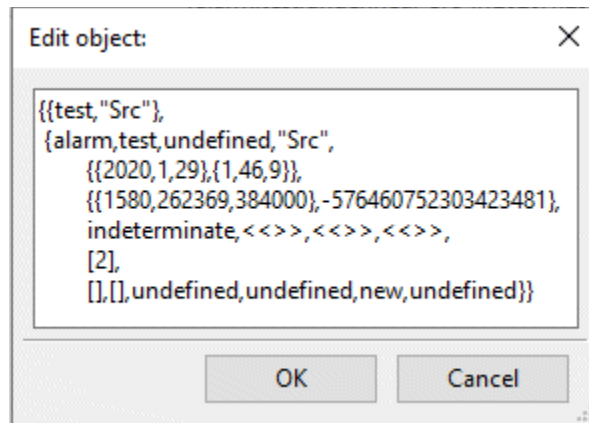


Figure 78 - chp02 Table Viewer alarmlist Edit Object dialog

43. Next, let's demonstrate the clearing of an existing alarm by entering "alarm:clear(test, "Src")." As shown in Figure 79.

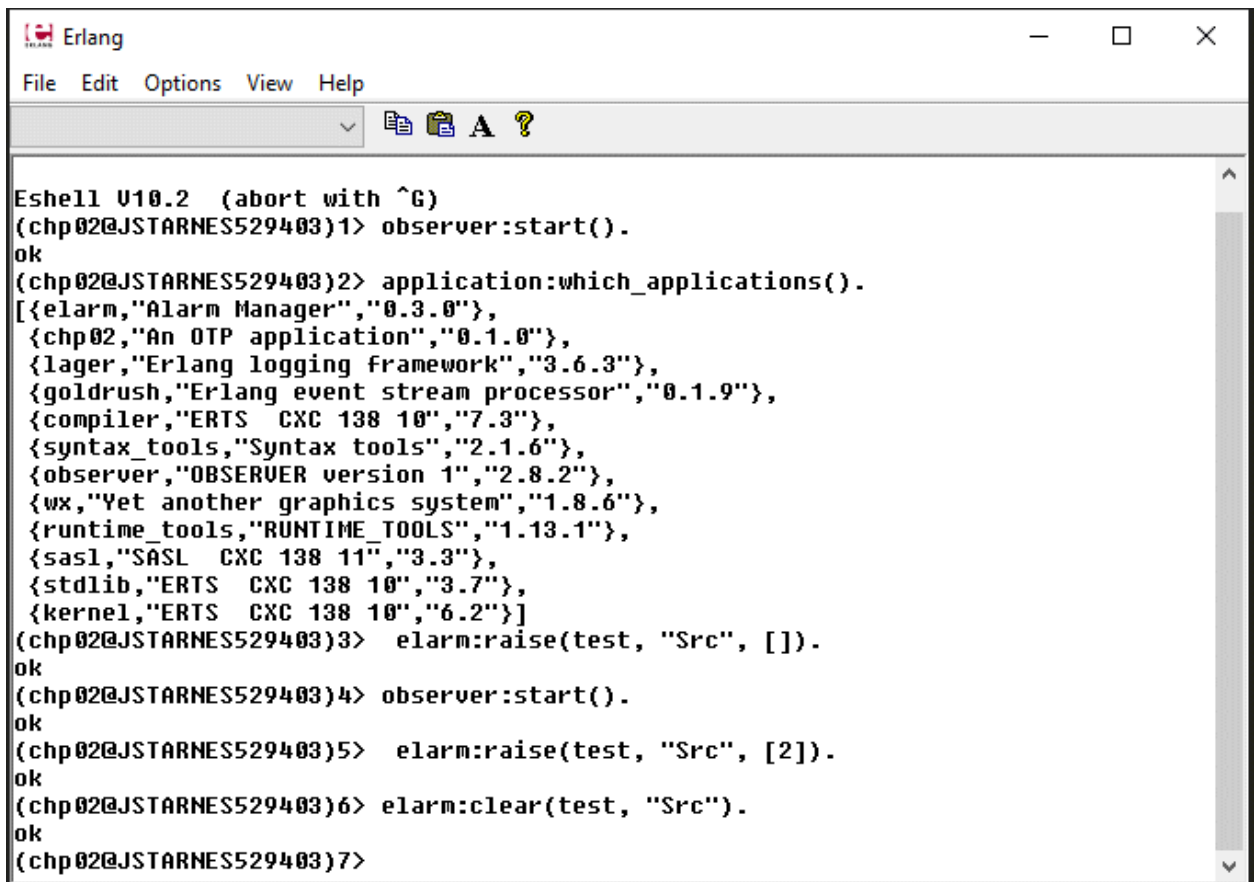


Figure 79 - chp02 clear alarm

44. Next, close any open TV views of the ets:alarmlist and then look at the contents of the ETS from the Table Viewer tab. The alarmlist and event_ids are empty again and the missing_alarms still contains a record of the missing configured alarm that existed at some point as show in Figure 80

Table Name	Objects	Size (kB)	Owner Pid	Owner Name	Table Id
alarmlist	0	2	<0.148.0>	elarm_server	#Ref<0.148.0>
configured_alarms	0	2	<0.148.0>	elarm_server	#Ref<0.148.0>
default	1	2	<0.76.0>	logger_std_h_default	#Ref<0.148.0>
event_ids	0	2	<0.148.0>	elarm_server	#Ref<0.148.0>
lager_config	3	3	<0.118.0>	lager_sup	
logger	7	13	<0.41.0>	logger	
missing_alarms	1	2	<0.148.0>	elarm_server	#Ref<0.148.0>
sasl	1	2	<0.84.0>	logger_std_h_sasl	#Ref<0.148.0>

Tables: 8

Figure 80 - chp02 Table Viewer alarms cleared

45. We covered a lot on the Elarm application and its features. In future chapter we will cover configured alarms and subscription to generated alarms for capture into an alarm.log which represents the history of raised and cleared alarms.
46. Next topic for the Erlang in Production is Metrics as discussed in page 8 of Jesper Louis Andersen slide deck in case you are still following along. He states that Metrics is a tool you can use to collect stats in determining how your system is currently behaving. We are going to include the Folsom library from Boundary which provides a metric system that you can instrument your code that aid you in situations where normal logging/tracing don't help with finding the proverbial smoking gun issue in your system.
47. The Folsom library supports the following six (6) types of metric: *counters*, *gauges*, *histograms* (and timers), *histories*, *meter_readers*, and *meters*. Let's take the hypothetical situation below and apply software metrics to aid us extracting the latent information that is hidden in a data stream.

In the Internet of Things (IoT) world you have a large body of remote sensors that your system monitors and each sensor is supposed to report its stats every 15 seconds. The department director approaches you and states that the customer is angry and claims our system isn't fulfilling our agreed upon service level agreement (SLA) for the data collection of remote sensors. The director has asked you to look into the issue and come up with a cost effective plan to resolve the issue.

The following is known:

- Over 10,000 sensors
- Distributed in 15 states
- IoT Units continuously monitor and report status every 15 seconds
 - PSI pressure readings
 - Temperature, Humidity

You start thinking and mentally wrestling with yourself and go through a list of thing to try.

- Look at logs? Not likely there are over 57 million reports daily.
- Send a field service technician to visit each device? Not likely, their union caps the service number to 10 devices a day, it takes one man over 3 man years to visit all of them and anyway it's too costly.

You decide to instrument your code with metrics and you remembered reading about **Spiral Meter** which counts the total numbers of reports in a sliding one minute window. You add the code to each receiver to perform the **Spiral Meter** for all 10,000 devices. The given update rate should average four (4) reports per minute. However, the IoT devices clocks aren't perfect so there will be fluctuations and a perfect four (4) reports per minute are not likely.

So far you collected some data, but you are struggling on the next step then you remembered that histograms can graphically depict the distribution of a population of readings. You decide to try the **Histogram Slide** by taking the results of **Spiral Meter**, which groups the units into different sample rates per minute which you display in Histogram chart like Figure 81 to quickly show the distribution of different sample rates, each bar represents a grouping of devices by update rates. Using the chart sourced from you metrics save the day as you mined your own data for found that particular subsets of IoT units are faulty due to a manufacturing error.

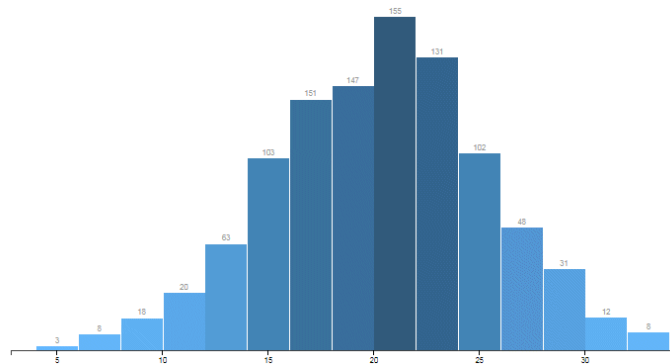
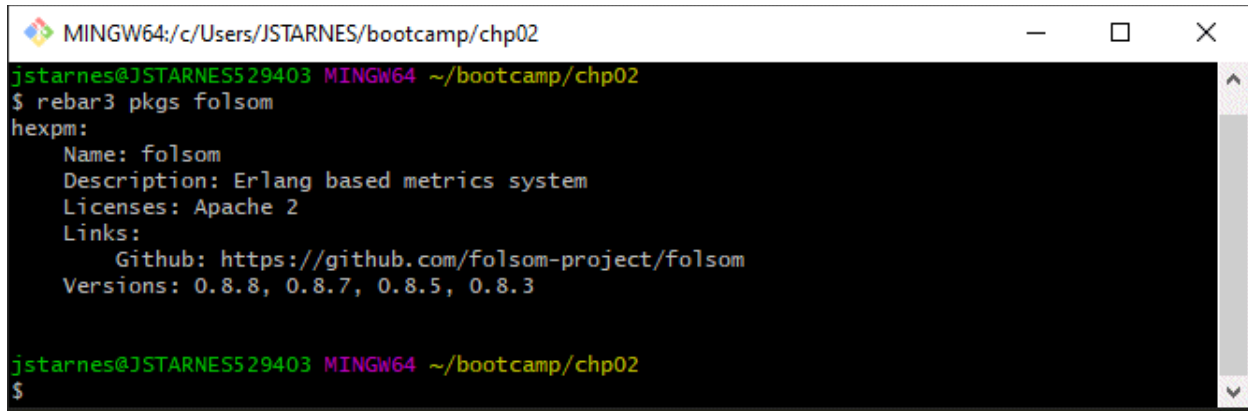


Figure 81 - Histogram Bar Chart

Hopefully, you get the idea that Metrics can provide solutions for situations where other approaches just don't help. Note the metrics aren't limited to software alone, the same tools could be used to extract business analytics, you just need to access to the right details.

48. Let's add the `folsom` package to `chp02 rebar.config` file and add the module the list of modules in our `relx` configuration section. First we need find the latest hex package of `folsom` by entering "`rebar3 pkgs folsom`" as shown in Figure 82. The results gives us the list of the versions, the `folsom` library has been around a while, so it's safe to say that the latest version hopefully aligns with our version of Erlang.



```

MINGW64:/c/Users/JSTARNES/bootcamp/chp02
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ rebar3 pkgs folsom
hexpm:
  Name: folsom
  Description: Erlang based metrics system
  Licenses: Apache 2
  Links:
    Github: https://github.com/folsom-project/folsom
  Versions: 0.8.8, 0.8.7, 0.8.5, 0.8.3

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$

```

Figure 82 - chp02 rebar3 pkgs folsom

49. Next, let's add the {folsom, 0.8.8} package and folsom to the relx configuration section of the chp02 rebar.config file as shown in Listing 15.

Listing 15 - chp02 rebar.config add folsom repo

```

{erl_opts, [debug_info, {parse_transform, lager_transform}]}.
{deps, [
  %% Packages
  {lager, "3.6.3"},
  {folsom, "0.8.8"},
  %% Source Dependencies
  {elarm, {git, "https://github.com/esl/elarm.git", {branch, "master"}}}
]}.

{relx, [{release, {chp02, "0.1.0"},
  [kernel, stdlib, sasl, runtime_tools,
   wx, observer, chp02, lager, elarm,
   folsom]},

  {sys_config, "./config/sys.config"},
  {vm_args, "./config/vm.args"},

  {dev_mode, false},
  {include_erts, false},
  {system_libs, false},
  {extended_start_script, true},
  {overlay, [{mkdir, "./log/sasl"}]}]}.

{profiles, [{prod, [{relx, [{dev_mode, false},
  {include_erts, true},
  {system_libs, true},
  {include_src, false},
  {debug_info, strip}]}]}]}.

```

50. Next, let's update the sys.config file with a place holder for configuration details for the folsom application as shown in Listing 16.

Listing 16 - chp02 sys.config add the folsom application configuration

```
[
  {chp02, []},
  %% kernel
  {kernel, [{logger_sasl_compatible, true}]},
  %% folsom config - [counter, gauge, histogram, history, meter, spiral, meter_reader]
  {folsom, []},
  %% SASL config
  {sasl, [
    {sasl_error_logger, {file, "log/sasl-error.log"}},
    {errlog_type, error},
    {error_logger_mf_dir, "log/sasl"},           % Log directory
    {error_logger_mf_maxbytes, 10485760},       % 10 MB max file size
    {error_logger_mf_maxfiles, 7}               % 7 files max
  ]},
  %% Lager config - [debug, info, notice, warning, error, critical, alert, emergency, none]
  {lager, [
    {log_root, "log"},
    {crash_log, "crash.log"},
    {crash_log_msg_size, 65536},
    {crash_log_size, 10485760},
    {crash_log_date, "$D0"},
    {crash_log_count, 7},
    {handlers, [
      {lager_file_backend,
        [{file, "error.log"}, {level, error}, {size, 10485760}, {date, "$D0"}, {count, 50}]},
      {lager_file_backend,
        [{file, "console.log"}, {level, info}, {size, 10485760}, {date, "$D0"}, {count, 50}]},
      {lager_file_backend,
        [{file, "debug.log"}, {level, debug}, {size, 10485760}, {date, "$D0"}, {count, 50}]}
    ]}
  ]}
].
```

51. Next, let's run the `rebar3 release` command as shown in Figure 83. Look, `rebar3` informed us that we are missing the `bear.app` from the kernel. This dependency is clearly from pulling in the `folsom` application, since that was the last dependency we added.

```

MINGW64:/c/Users/JSTARNES/bootcamp/chp02
r-0.8.7.tar is up to date, reusing it
==> Fetching goldrush v0.1.9
==> Version cached at c:/CIS/source/CORE/pgdmtest/.cache/hex/hexpm/packages/goldrush-0.1.9.tar is up to date, reusing it
==> Compiling bear
==> "c:/Users/JSTARNES/bootcamp/chp02/_build/default/lib/bear/ebin/bear.app" is missing kernel from applications list
==> "c:/Users/JSTARNES/bootcamp/chp02/_build/default/lib/bear/ebin/bear.app" is missing stdlib from applications list
==> Compiling folsom
==> Compiling elarm
==> Compiling goldrush
==> Compiling lager
==> Compiling chp02
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
      c:/Users/JSTARNES/bootcamp/chp02/_build/default/lib
      c:/Users/JSTARNES/bootcamp/chp02/apps
      c:/Program Files/erl10.2/lib
==> Resolved chp02-0.1.0
==> release successfully created!

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$

```

Figure 83 - chp02 rebar3 release missing from kernel

Next, let's fix the dependency issue by modifying the rebar.config relx section and add the bear prior to the folsom entry. The order technically is not important but it's a good practice to keep items in order of their dependencies.

52. Listing 17 below shows the correction for the missing bear dependency.

Listing 17 - chp02 rebar.config add bear dependency

```

{erl_opts, [debug_info, {parse_transform, lager_transform}]}.
{deps, [
  %% Packages
  {lager, "3.6.3"},
  {folsom, "0.8.8"},
  %% Source Dependencies
  {elarm, {git, "https://github.com/esl/elarm.git", {branch, "master"}}}
]}.

{relx, [{release, {chp02, "0.1.0"},
  [kernel, stdlib, sasl, runtime_tools,
   Wx, observer, chp02, lager, elarm,
   bear, folsom]},

  {sys_config, "./config/sys.config"},
  {vm_args, "./config/vm.args"},

  {dev_mode, false},
  {include_erts, false},
  {system_libs, false},
  {extended_start_script, true},
  {overlay, [{mkdir, "./log/sasl"}]}]}.

{profiles, [{prod, [{relx, [{dev_mode, false},
  {include_erts, true},
  {system_libs, true},

```

```

        {include_src, false},
        {debug_info, strip}]]]
    }].

```

53. Next, let's modify the `chp02_app.erl` to make sure the `folsom` application is started as shown in Figure 84.

```

1  %/0/0/-----
2  %/0/ @doc chp02 public API
3  %/0/ @end
4  %/0/-----
5
6  -module(chp02_app).
7
8  -behaviour(application).
9
10 -export([start/2, stop/1]).
11
12 start(_StartType, _StartArgs) ->
13     application:start(sasl),
14     application:ensure_all_started(lager),
15     application:ensure_all_started(elarm),
16     application:ensure_all_started(folsom),
17     chp02_sup:start_link().
18
19 stop(_State) ->
20     ok.
21
22 %% internal functions
23

```

Figure 84 - `chp02_app.erl` start `folsom` application

54. Next, let's do a `rebar3` release followed by `./chp.cmd` console command. Now go in the console window and let's create a new counter and do the following sequence of commands:
- A. `folsom_metrics:new_counter(device).`
 - B. `folsom_metrics:notify({device, {inc, 1}}).`
 - C. `folsom_metrics:get_metric_value(device).`
 - D. `folsom_metrics:notify({device, {inc, 6}}).`
 - E. `folsom_metrics:get_metric_value(device).`
 - F. `folsom_vm_metrics:get_memory().`
55. Note the `folsom_vm_metrics:get_memory()`, which provides memory details for the Erlang VM. I left a lot of other calls out and urge you to read up on them at: <https://github.com/boundary/folsom>.

```

Erlang
File Edit Options View Help
(chp02@JSTARNES529403)4> folsom_metrics:new_counter(device).
ok
(chp02@JSTARNES529403)5> folsom_metrics:notify({device, {inc, 1}}).
ok
(chp02@JSTARNES529403)6> folsom_metrics:get_metric_value(device).
1
(chp02@JSTARNES529403)7> folsom_metrics:notify({device, {inc, 6}}).
ok
(chp02@JSTARNES529403)8> folsom_metrics:get_metric_value(device).
7
(chp02@JSTARNES529403)9>

```

Figure 85 - chp02 folsom metrics

56. Few last thoughts, the folsom application uses the Erlang Term Storage (ETS) tables just like the alarm application and the data does not persist to the disk. Also, the folsom application allows for the upfront configuration of the named metrics in your application's sys.config file {folsom, []} tuple.
57. Next topic for the Erlang in Production is Tracing/Recon as discussed in page 11 of Jesper Louis Andersen slide deck and refers to capability of Erlang to trace programs and the [Recon](#) library contribution by fellow Erlang evangelist Fred Herbert. One of the first lessons a programmer learns when working with Erlang OTP is debugging Erlang code is not very practical as debugger interrupts the processing of data and once the process needs to interact with other Erlang processes you are debugging, its call start timing out and crashing, possibly taking down the entire node with it. Tracing provides a mechanism to capture details that does not interfere with the program's execution and still gives you all the relevant data you need.
58. Next, let's add the recon library by repeating the process of finding which Hex package is the latest using the "rebar3 pkgs recon" similar to steps we did before for the folsom library as show in Figure 86.

```

MINGW64:/c:/Users/JSTARNES/bootcamp/chp02
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ rebar3 pkgs recon
hexpm:
  Name: recon
  Description: Diagnostic tools for production use
  Licenses: BSD
  Links:
    Documentation: http://ferd.github.io/recon/
    Github: https://github.com/ferd/recon/
  Versions: 2.5.0, 2.4.0, 2.3.6, 2.3.5, 2.3.4, 2.3.3, 2.3.2, 2.3.1, 2.3.0, 2.2.1, 2.2.0

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ |

```

Figure 86 - chp02 rebar3 pkgs recon

59. Next, let's add the {folsom, 0.8.8} package and folsom to relx configuration section of the chp02 rebar.config file as shown in Figure 86.

Listing 18 – chp02 rebar3.config add recon repo

```
{erl_opts, [debug_info, {parse_transform, lager_transform}]}.
{deps, [
  %% Packages
  {lager, "3.6.3"},
  {folsom, "0.8.8"},
  {recon, "2.5.0"},
  %% Source Dependencies
  {elarm, {git, "https://github.com/esl/elarm.git", {branch, "master"}}}
]}.

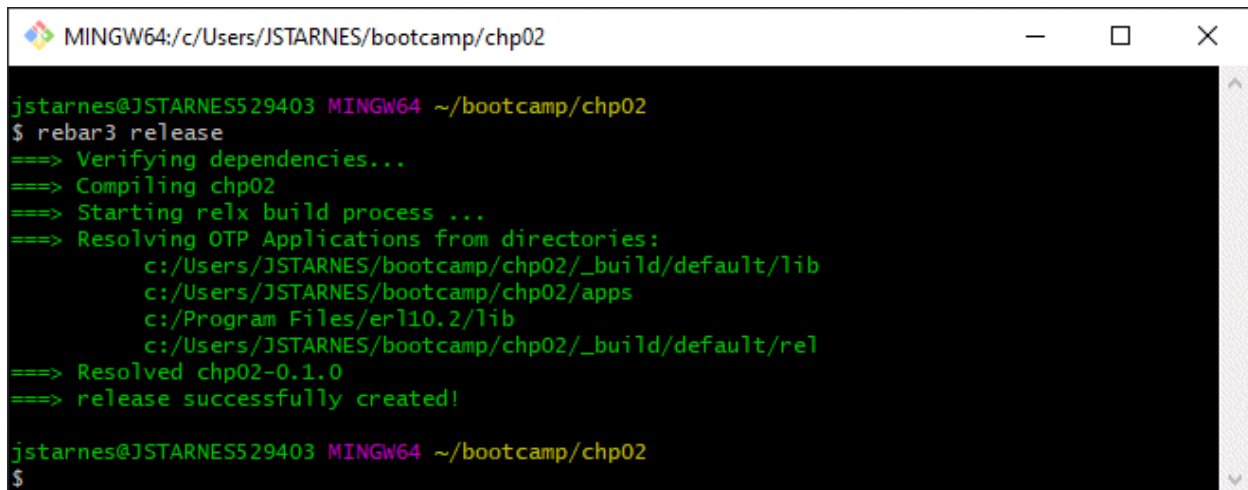
{relx, [{release, {chp02, "0.1.0"},
  [kernel, stdlib, sasl, runtime_tools,
   Wx, observer, chp02, lager, elarm,
   folsom, recon]},

  {sys_config, "./config/sys.config"},
  {vm_args, "./config/vm.args"},

  {dev_mode, false},
  {include_erts, false},
  {system_libs, false},
  {extended_start_script, true},
  {overlay, [{mkdir, "./log/sasl"}]}]}.

{profiles, [{prod, [{relx, [{dev_mode, false},
  {include_erts, true},
  {system_libs, true},
  {include_src, false},
  {debug_info, strip}]}]}]}.
}
```

60. Next, let's run the rebar3 release for chp02 as show in Figure 87.



```
MINGW64:/c:/Users/JSTARNES/bootcamp/chp02
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ rebar3 release
==> Verifying dependencies...
==> Compiling chp02
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
    c:/Users/JSTARNES/bootcamp/chp02/_build/default/lib
    c:/Users/JSTARNES/bootcamp/chp02/apps
    c:/Program Files/erl10.2/lib
    c:/Users/JSTARNES/bootcamp/chp02/_build/default/rel
==> Resolved chp02-0.1.0
==> release successfully created!

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$
```

Figure 87 - cho02 rebar3 release

61. Before we discuss tracing, it's important to note that the Recon application is a collection of a number of utilities all designed to help diagnose issues in a production deployment of an Erlang OTP application, the following is a list of the included modules:

- **recon** - Main module, contains basic functionality to interact with the recon application. It includes functions to gather information about processes and the general state of the virtual machine, ports, and OTP behaviours running in the node. It also includes a few functions to facilitate RPC calls with distributed Erlang nodes.
 - **recon_alloc** - Regroups functions to deal with Erlang's memory allocators, or particularly, to try to present the allocator data in a way that makes it simpler to discover the presence of possible problems.
 - **recon_lib** - Regroups useful functionality used by recon when dealing with data from the node. Would be an interesting place to look if you were looking to extend Recon's functionality.
 - **recon_map** - This module handles formatting maps. It allows for trimming output to selected fields, or to nothing at all. It also adds a label to a printout. To set up a limit for a map, you need to give recon a way to tell the map you want to trim from all the other maps, so you have to provide something like a 'type definition'. It can be either another map which is compared to the arg, or a fun.
 - **recon_rec** - An extension to recon_trace, enables printing out records in a more readable format based on known record definitions.
 - **recon_trace** - Provides production-safe tracing facilities, to dig into the execution of programs and function calls as they are running.
62. The Erlang Trace BIFs (Binary Internal Functions) allow for the tracing of Erlang code. The tracing works by specifying the PID specifications and trace patterns. The PID specifications limit what processes to target, whereas the trace patterns limit which functions. The trace patterns for functions are specified in two-parts: specifying by modules, functions, and arity, and then with Erlang match specifications². The intersection between Matching PIDS and Matching Trace Patterns is what ultimately gets traced as shown in the Tracing Venn diagram in Figure 88.

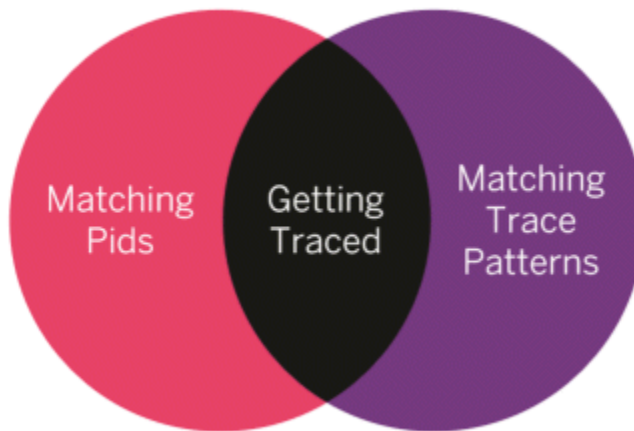


Figure 88 - Tracing Venn diagram

63. The Erlang Trace BIFs can be found at [trace-3](#), the recon_trace library is much easier to use and safer as it has limiters that protect against dumb decisions (matching all calls on a node being trace, for example). Let's try a few calls, but to be honest it's premature to do any tracing at this point in the Erlang Bootcamp as we haven't added code of our own so far. Start the chp02 application in a console and go to the console and type the following command as shown in **Error! Reference source not found.**

² http://www.erlang.org/apps/erts/match_spec.html

```

Erlang
File Edit Options View Help
Erlang/OTP 21 [erts-10.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:30]
Eshell U10.2 (abort with ^G)
(chp02@JSTARNES529403)1> recon_trace:calls({queue, '_', fun(_) -> return_trace()
end}, 3).
32
12:54:43.238000 <0.67.0> queue:peek({[],[]})
12:54:43.238000 <0.67.0> queue:peek/1 --> empty
12:54:43.238000 <0.67.0> queue:in({<0.69.0>,
{put_chars_sync,unicode,<<"\n">>,
{{<0.160.0>,#Ref<0.2290181822.2149580803.133237>},ok}}}, {[],[]
})
Recon tracer rate limit tripped.
(chp02@JSTARNES529403)2>

```

Figure 89 - recon_trace calls

64. Next, let's close the chp02 console application down by entering a CTRL+G followed by a 'q' character and return. Next, let's run the "rebar3 tree" command to show all the dependencies of the chp02 application as shown in Figure 90.

```

MINGW64:/c:/Users/JSTARNES/bootcamp/chp02
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$ rebar3 tree
==> Verifying dependencies...
├─ chp02-0.1.0 (project app)
│   ├── elarm-0.3.0 (git repo)
│   ├── folsom-0.8.8 (hex package)
│   │   └─ bear-0.8.7 (hex package)
│   ├── lager-3.6.3 (hex package)
│   │   └─ goldrush-0.1.9 (hex package)
│   └─ recon-2.5.0 (hex package)
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp02
$

```

Figure 90 - chp02 rebar3 tree

Summary

We covered important topics and created a hypothetical basis application for the Erlang Bootcamp, we accomplished the following items:

- Added the Erlang System Architecture Support Libraries (SASL) for adding alarm_handler, release_handler, systools
- Learn in rebar.config that overlays are command such as: mkdir, anmd cp. That aids in setting target release folders in the relx configuration section
- Created the {mkdir, "/log/sasl"} directory using the overlay command
- Added the lager application extending the Erlang logger functionality with log rotation and crash dump files

- Added the `{logger_sasl_compatible, true}` to the kernel config so to allow the lager library working with version OTP 21.0 or greater to integrate with Erlang OTP logging facility.
- Added the lager dependency package to deps and relx configuration sections of the `chp02` application's `rebar.config`
- Added the lager configuration for `crash.log` and rotation of `error.log`, `console.log`, and `debug.log` in `chp02` `sys.config` file
- We learned about crash logs and how they can help identify issues by providing stacktrace and relevant data at the time of the crash
- We learned that while crashes can happen the Erlang OTP application as a whole can remain operational
- We learned while adding new dependencies that other sub dependencies may occur and need to be added to relx section of application's `rebar.config` file
- We learned that for each new application dependency added we need to add calls in startup of application to start those new application by using the `application:start` or `application:ensure_all_started` calls.
- We reviewed the log files subdirectory underneath the `chp02` application listing of log files: *console.log*, *crash.log*, *debug.log*, *error.log*, and *sasl-error.log*
- We learned about lager's logging levels and associated lager functions for those levels
- We added a sample `lager:info(...)` call and reviewed the contents of the `console.log` and noticed other dependency application logs already captured by lager in log file along with our sample data
- We learned that it is important not to go overboard with logging, and the lager is for general application logging, and other types of logging exist that are better suited for application specific content
- We learned about the SASL report logs and how they provide: *Supervisory*, *Progress*, and *Crash* reports.
- We learned that SASL reports require usage of Report Browser module aka `rb`, and that the SASL reports in certain circumstances be your best choice find startup issue via Progress Reports, and Supervisory Reports for unexpected termination of child processes, and Crash Reports for unexpected process crashes for processes started with `proc_lib:spawn` and `proc_lib:spawn_link`.
- Added the `elarm` application as a dependency for the `chp02` application using `https` git commands for repo not using hex package manager.
- We learned about the observer application and reviewed: *System*, *Application*, and *Table Viewer* tabs
- We learned that `elarm` application uses Erlang Term Storage (ETS) and creates the following tables: *alarmlist*, *configured_alarms*, *event_ids*, and *missing_alarms*
- We learned that periodically it is important to run a production release to make sure all dependencies are satisfied for the production release
- We learned more about the observer application by reviewing the Processes tab and learned that the `MsgQ` count represents the number of messages waiting to be processed by the process and checking this value can inform you on a possible blockage with your process.
- We learned about the `application:which_applications()` call that shows what applications are currently running from the console, shell or application
- We learned how to raise/clear alarms and to add relevant associated data to an alarm that can be used later in an report.
- We learned how to use the observer application to review the contents of alarm entry in the `alarmlist` using Table Viewer and Edit object dialogs
- We learned that alarms represent current state of application and `alarms.log` represent the history of alarms, first creates, updated, and finally cleared.
- Added the `folsom` application that provides Metrics to our to the `chp02` application
- We learned that the `folsom` application supports the following types of metrics: *counters*, *gauges*, *histograms* (and timers), *histories*, *meter_readers*, and *meters*
- We explored a hypothetical situation of an IoT company using metrics to diagnose performance issues with a large body of remote sensors.

- We learned that how to use the folsom application metric functions to instrument our application to capture key performance metrics and how you can compose different metric functions are part of a solution.
- We learned that Metrics addresses problems that cannot be resolved by other means easily due to the high volume of data and how Metrics help mine latent information from an application's data stream
- We learned about the importance of tracing and the the Recon application which provides safer tools for use in an production enviornment
- We learned how to get a list of dependencies using the rebar3 tree command

We covered a lot of material in this chapter, but kept the details to a breif introduction as many of the tools described only show their value with the inclusion of the our application code. The underlying goal of this chapter was to establish a baseline for our hypothetical application. Now that we covered most of the typically supporting applications, we can formalize this with a basis project template that in a single command line "rebar new basis <appname>" will add all of the above applications.

CHAPTER 3 - ERLANG OTP CLUSTERS

1. In Chapter 3 we will learn about Erlang OTP Clusters and we will build different two (2) different types of clusters. Before we can get started we need to update the templates stored in your `~/.config/rebar3/templates` folder. Below is a summary of templates that have been updated as a result of work in Chapters 1 thru 3:
 - a. First set of templates for Chapter 1
 - b. Second set of templates added basis project and the end that chapter that captures all the dependency applications discussed in Chapter 2. Now with a single rebar2 new basis chp02 builds the hypothetical application basis starting point.
 - c. Third set of templates added basis_cluster and basis_cluser_failover templates for Chapter 3, which includes Makefile usage and builds a three(3) node cluster for develop release testing.
2. You will need to go to Section 3.3 GNU Make for Windows and follow the instructions for adding the GNU Make to the environment so we can take advantage of Makefiles.
3. Go to the 'G:\SETS\Jeff\config\rebar3\templates\chp03' and copy the rebar3_templates_chp03.zip to your local drive and extract the contents into your local "`~/.config/rebar3/templates`" directory. Your directory "`~/.config/rebar3/templates`" should look similar to Figure 91.

```

MINGW64:/c/Users/JSTARNES/\.config/rebar3/templates
jstarnes@JSTARNES529403 MINGW64 ~/.config/rebar3/templates
$ cd ~/.config/rebar3/templates

jstarnes@JSTARNES529403 MINGW64 ~/.config/rebar3/templates
$ ls
app.erl                escript_rebar.config  plugin_README.md
app.template           gen_event.erl         provider.erl
app_rebar.config       gen_event.template    README.md
basis/                 gen_server.erl        readme.txt
basis.template         gen_server.template   rebar.config
basis_cluster/         gen_statem.erl        release.template
basis_cluster.template gen_statem.template   relx_rebar.config
basis_cluster_failover/ gitignore              report.cmd
basis_cluster_failover.template lib.template          sup.erl
cmake.template         LICENSE               sys.config
command.cmd            Makefile              sys.config.src
commonest.erl          mod.erl               umbrella.template
commonest.template     otp_app.app.src       vm.args
escript.template       otp_lib.app.src       vm.args.src
escript_mod.erl        plugin.erl             vm.zip
escript_README.md      plugin.template

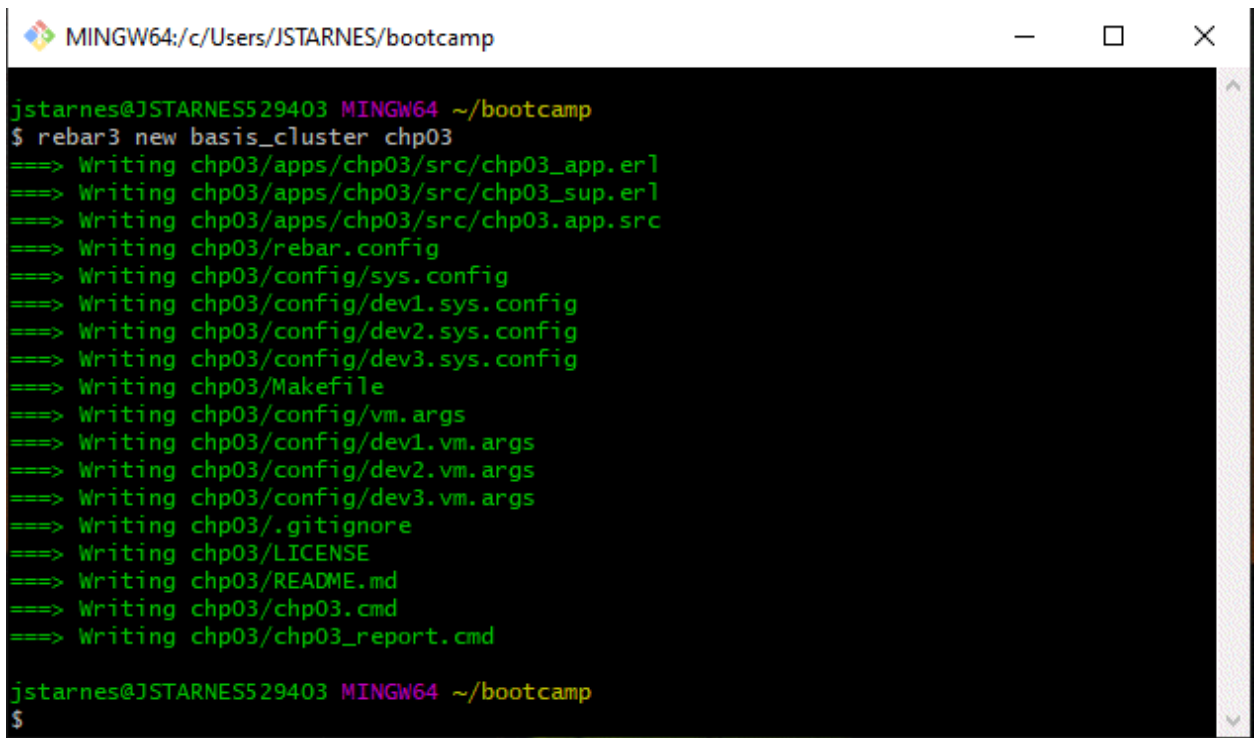
```

Figure 91 - listing of the `~/.config/rebar3/templates` directory

4. Before we dive into Erlang OTP Clusters, I'd like share an observation you may not have considered. The observation I am speaking of is the trend that back office systems should be run underneath a Virtual Stack such as VMware, Docker, or Kubernetes, etc. It seems that some IT groups view VMware as the solution for all their routine backup/restore woes. Moreover, companies that have a need to make their solutions run in a clustered environment and perhaps have the need to have High Availability and/or Fault Tolerance bolted on to their existing applications. Others decide that restructuring their application to run in a containerized environment to achieve their goals. It's ironic, so many companies are doling out large sums of

money to VMware or RedHat OpenShift for the privilege having their application become quasi-clustered and become HA or FT as a result. The downside of going in that direction is the additional cost and potential limitations that the VMware and other solutions put on the hosted applications may not suit your application's needs. The VMware approach is emulating an entire OS, not just your application. Docker/Kubernetes containerization solutions require restructuring your software, either way it seems that companies are trying to adapt their application and are lured into these costly solutions for a potential quick turn-around. Don't get me wrong, it's really neat to be able to spin up a VM to run a piece of SW you need. That being said, the Erlang OTP Clusters are supported at the application level and only focus on application concerns.

5. Let's get started going to your `~/bootcamp` directory and running the `"rebar3 new basis_cluster chp03"` command as shown in Figure 92.



```
MINGW64:/c/Users/JSTARNES/bootcamp

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp
$ rebar3 new basis_cluster chp03
==> Writing chp03/apps/chp03/src/chp03_app.erl
==> Writing chp03/apps/chp03/src/chp03_sup.erl
==> Writing chp03/apps/chp03/src/chp03_app.src
==> Writing chp03/rebar.config
==> Writing chp03/config/sys.config
==> Writing chp03/config/dev1.sys.config
==> Writing chp03/config/dev2.sys.config
==> Writing chp03/config/dev3.sys.config
==> Writing chp03/Makefile
==> Writing chp03/config/vm.args
==> Writing chp03/config/dev1.vm.args
==> Writing chp03/config/dev2.vm.args
==> Writing chp03/config/dev3.vm.args
==> Writing chp03/.gitignore
==> Writing chp03/LICENSE
==> Writing chp03/README.md
==> Writing chp03/chp03.cmd
==> Writing chp03/chp03_report.cmd

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp
$
```

Figure 92 - `rebar3 new basis_cluster chp03` command

6. Next, change your directory to `chp03` and run the `"make devrel"` command that builds the three (3) cluster members: `chp03_dev1`, `chp03_dev2`, and `chp03_dev3` as shown in Figure 93.

```

MINGW64:/c/Users/JSTARNES/bootcamp/chp03
c:/Users/JSTARNES/bootcamp/chp03/_build/dev2/lib
c:/Users/JSTARNES/bootcamp/chp03/apps
c:/Program Files/erl10.2/lib
==> Resolved chp03-0.1.0
==> release successfully created!
rebar3 as dev3 release
==> Verifying dependencies...
==> Linking _build/default/lib/bear to _build/dev3/lib/bear
==> Linking _build/default/lib/elarm to _build/dev3/lib/elarm
==> Linking _build/default/lib/folsom to _build/dev3/lib/folsom
==> Linking _build/default/lib/lager to _build/dev3/lib/lager
==> Linking _build/default/lib/recon to _build/dev3/lib/recon
==> Linking _build/default/lib/goldrush to _build/dev3/lib/goldrush
==> Compiling chp03
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
    c:/Users/JSTARNES/bootcamp/chp03/_build/dev3/lib
    c:/Users/JSTARNES/bootcamp/chp03/apps
    c:/Program Files/erl10.2/lib
==> Resolved chp03-0.1.0
==> release successfully created!

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp03
$

```

Figure 93 - make devrel command

- Next, let's run the make dev1-console command and press return, then run make dev2-console and press return, then run make dev3-console command and press return.
- Next, move the console windows around so as to be able to see the Git Bash shell window clearly and the chp03_dev1 chp03_dev2, and chp03_dev3 as shown in Figure 94.

```

MINGW64:/c/Users/JSTARNES/bootcamp/chp03

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp03
$ make dev1-console
"/c/Users/JSTARNES/bootcamp/chp03/_build/dev1/rel/chp03/bin/chp03.cmd" console

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp03
$ make dev2-console
"/c/Users/JSTARNES/bootcamp/chp03/_build/dev2/rel/chp03/bin/chp03.cmd" console

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp03
$ make dev3-console
"/c/Users/JSTARNES/bootcamp/chp03/_build/dev3/rel/chp03/bin/chp03.cmd" console

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp03
$

```

Figure 94 – make dev1-console, make dev2-console, make dev3-console

- Next, run the "nodes()." command in each console window, they should all return an empty list, aka []. Currently right now each application is unaware of the other applications. These nodes need to be joined by performing a net_kernel:connect(Node) or a net_adm:pings(Node). Let's

connect all three nodes by using the `net_kernel:connect('chp03_dev2@<NodeName>')` and `net_kernel:connect('chp03_dev3@<NodeName>')` from the `chp03_dev1@<NodeName>` console as show in Figure 95.

```

Erlang
File Edit Options View Help

Erlang/OTP 21 [erts-10.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:30]
Eshell V10.2 (abort with ^G)
(chp03_dev1@JSTARNES529403)1> net_kernel:connect_node('chp03_dev2@JSTARNES529403').
true
(chp03_dev1@JSTARNES529403)2> net_kernel:connect_node('chp03_dev3@JSTARNES529403').
true
(chp03_dev1@JSTARNES529403)3>

```

Figure 95 - `net_kernel:connect_node` calls to `chp03_dev2` and `chp03_dev3`

10. Next, let's verify that all three nodes are aware of each other by executing the `nodes()` function in all three nodes. On each node the list is the other two servers in the cluster as seen in Table 3.

Table 3 - `nodes()` results on all three nodes

Node Name	<code>nodes()</code> result
<code>chp03_dev1</code>	<code>[chp03_dev2@JSTARNES529403,chp03_dev3@JSTARNES529403]</code>
<code>chp03_dev2</code>	<code>[chp03_dev1@JSTARNES529403,chp03_dev3@JSTARNES529403]</code>
<code>chp03_dev3</code>	<code>[chp03_dev1@JSTARNES529403,chp03_dev2@JSTARNES529403]</code>

11. Note, typically on any node you may want the list of all participating members, this is typically done by performing by combining the `node()` function call and the `nodes()` function calls in a list with a append pipe `|` character as shown in Figure 96.

```

Erlang
File Edit Options View Help

Erlang/OTP 21 [erts-10.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:30]
Eshell V10.2 (abort with ^G)
(chp03_dev2@JSTARNES529403)1> [node() | nodes()].
[chp03_dev2@JSTARNES529403,chp03_dev1@JSTARNES529403,
 chp03_dev3@JSTARNES529403]
(chp03_dev2@JSTARNES529403)2>

```

Figure 96 - `[node() | nodes()]` list append

12. Before we go any further, the `"rebar3 new basis_cluster chp03"` command generates a unique `vm.args` per node file used to hold the defined `--sname` and secret cookie value that is used between the `dev1`, `dev2`, and `dev3` cluster members. Let's perform a `cat` command on all three `vm.args` file for `dev1`, `dev2`, and `dev3` as shown in Figure 97. The `vm.args` config file contains the

–sname parameter and the –setcookie value that allows all three nodes to communicate with each other.

```

MINGW64: c:/Users/JSTARNES/bootcamp/chp03
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp03
$ cat ./_build/dev1/rel/chp03/releases/0.1.0/vm.args
-sname chp03_dev1

-setcookie 8812

+K true
+A30

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp03
$ cat ./_build/dev2/rel/chp03/releases/0.1.0/vm.args
-sname chp03_dev2

-setcookie 8812

+K true
+A30

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp03
$ cat ./_build/dev3/rel/chp03/releases/0.1.0/vm.args
-sname chp03_dev3

-setcookie 8812

+K true
+A30

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp03
$

```

Figure 97 - cat command on all three dev releases

The –sname and –name parameter are used to make the Erlang node distributed and the sname is the short name for the node Name@host, where Name is not a fully qualified. This parameter does not rely on a DNS server, whereas the –name parameter does have a fully qualified Name and relies upon the DNS server. The –setcookie value must match for each node participating in the cluster.

13. The Erlang OTP Cluster we just created is a simple load balancing kind of cluster, where all three members have the exact same code, although they don't have to be identical and no single server is the master of the cluster. So, what is this type of cluster good for? Basically, the load-balancing cluster is typically of a web server farm or other kinds of services where you need to distribute the load across all members of the cluster. Typically, a reverse proxy like HAProxy or NGINX is used to distribute the load by acting as a go-between for the clients and the clustered servers back end.
14. Now the servers are clustered, your application can take advantage of resources across the cluster making request on a remote cluster member by either remote procedure calls (RPC), messages or some other communication method such as queues, TCP/IP sockets, etc. Perhaps your application has the need to coordinate request between nodes for example a workflow, etc. Let's try a RPC call from chp03_dev2 console to the chp03_dev3@<NodeName> as shown in Figure 98.



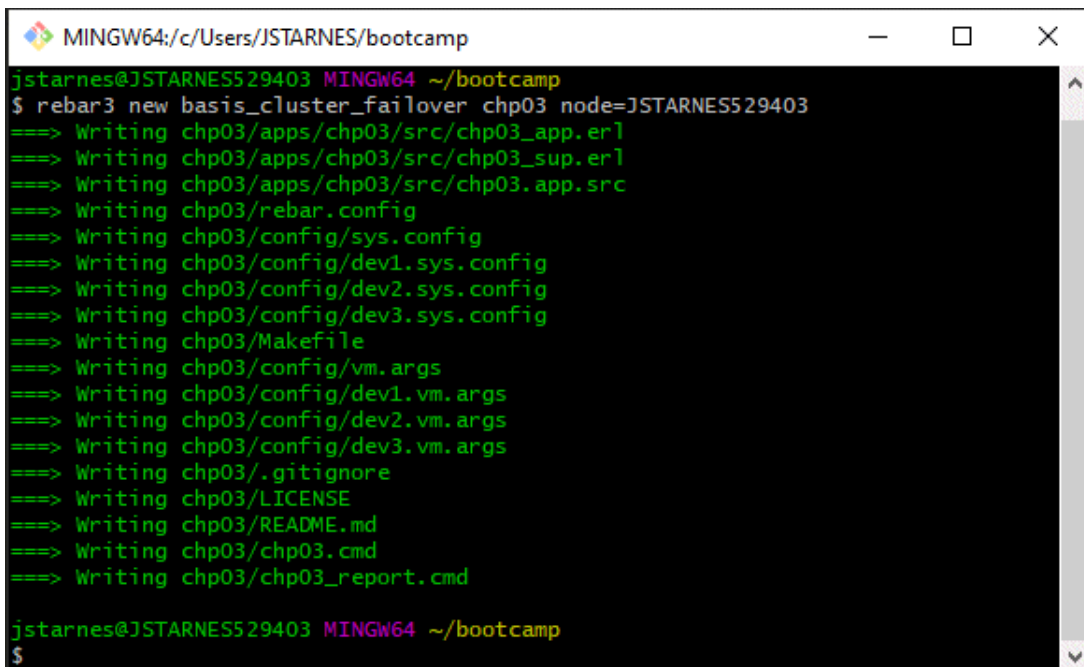
```

Eshell U10.2 (abort with ^G)
(chp03_dev2@JSTARNES529403)1> [node() | nodes()].
[chp03_dev2@JSTARNES529403,chp03_dev1@JSTARNES529403,
 chp03_dev3@JSTARNES529403]
(chp03_dev2@JSTARNES529403)2> rpc:call('chp03_dev3@JSTARNES529403', erlang, node, []).
chp03_dev3@JSTARNES529403
(chp03_dev2@JSTARNES529403)3>

```

Figure 98 - RPC call from chp03_dev2 to chp03_dev3

15. Next, let's explore a different kind Erlang OTP Cluster, this type is known in Erlang documentation as a Distributed Application³ or better known as a failover cluster. This type of cluster support a single active application and one or more standby nodes. This type of cluster supports both automatic failover and also manual takeover capabilities.
16. Let's get started, but first we need to get out of all three (3) chp03 dev1, dev2, and dev3 consoles. Next, go to your Git Bash shell and change directory to "~/bootcamp" folder and run the Windows "TASKKILL //F //IM epmd.exe" command to kill the Erlang port mapper daemon that locks your "~/bootcomap/chp03" directory preventing you the ability to purge the entire contents. Next, run the "rm -fr chp03" command from your "~/bootcamp" directory to remove the contents of chp03 that we created with the basis_cluster template⁴.
17. Next, let's run the "rebar3 new basis_cluster_failover chp03" command in your "~/bootcamp" directory as shown in Figure 99.



```

MINGW64:/c/Users/JSTARNES/bootcamp
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp
$ rebar3 new basis_cluster_failover chp03 node=JSTARNES529403
==> Writing chp03/apps/chp03/src/chp03_app.erl
==> Writing chp03/apps/chp03/src/chp03_sup.erl
==> Writing chp03/apps/chp03/src/chp03_app.src
==> Writing chp03/rebar.config
==> Writing chp03/config/sys.config
==> Writing chp03/config/dev1.sys.config
==> Writing chp03/config/dev2.sys.config
==> Writing chp03/config/dev3.sys.config
==> Writing chp03/Makefile
==> Writing chp03/config/vm.args
==> Writing chp03/config/dev1.vm.args
==> Writing chp03/config/dev2.vm.args
==> Writing chp03/config/dev3.vm.args
==> Writing chp03/.gitignore
==> Writing chp03/LICENSE
==> Writing chp03/README.md
==> Writing chp03/chp03.cmd
==> Writing chp03/chp03_report.cmd

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp
$

```

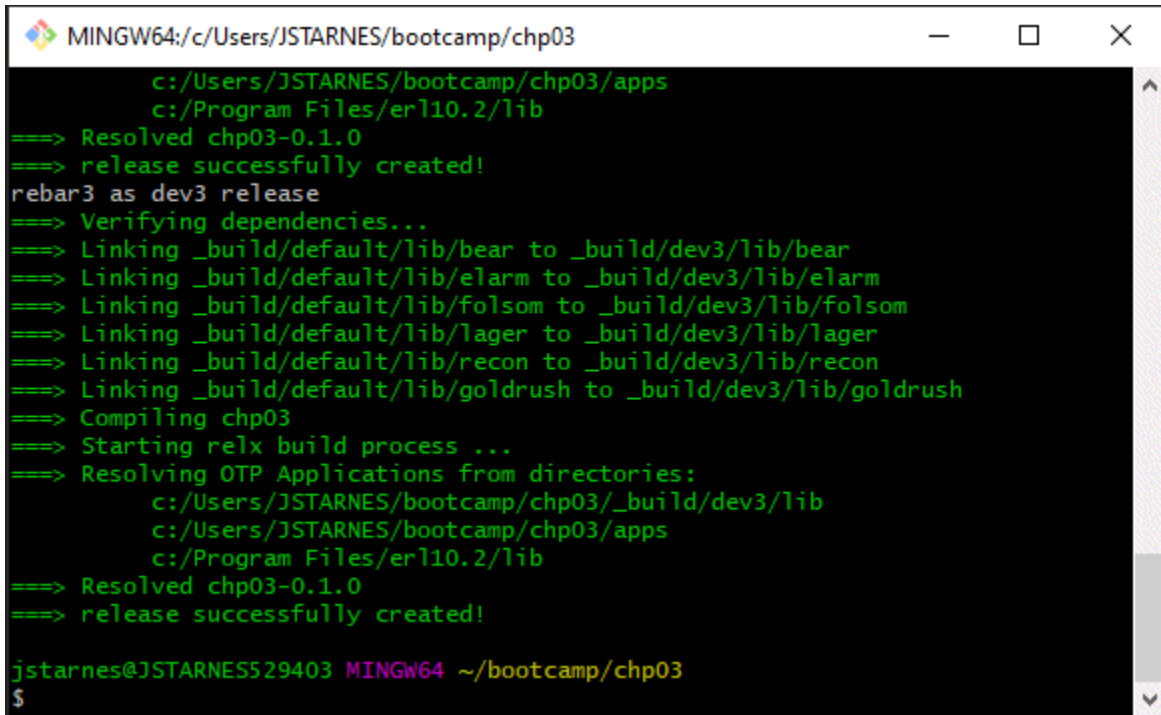
Figure 99 - rebar3 new basis_cluster_failover chp03 command

³ Distributed Application -

https://erlang.org/doc/design_principles/distributed_applications.html#specifying-distributed-applications

⁴ Note the Erlang port mapper daemon was created as a result of your testing the basis_cluster.

18. Next, change your directory to the chp03 folder just created by the basis_cluster_failover template. Let's run the "make devrel" command as before to generate the three (3) development nodes in our failover cluster as shown in Figure 100.



```

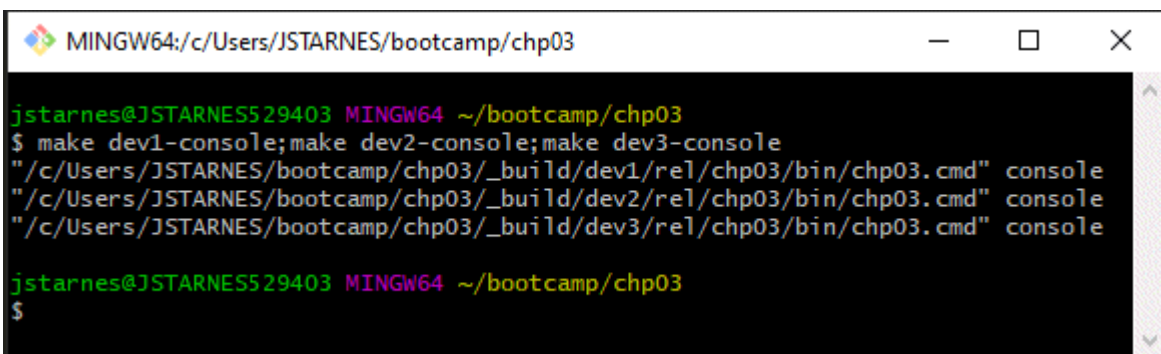
MINGW64:/c:/Users/JSTARNES/bootcamp/chp03
c:/Users/JSTARNES/bootcamp/chp03/apps
c:/Program Files/erl10.2/lib
==> Resolved chp03-0.1.0
==> release successfully created!
rebar3 as dev3 release
==> Verifying dependencies...
==> Linking _build/default/lib/bear to _build/dev3/lib/bear
==> Linking _build/default/lib/elarm to _build/dev3/lib/elarm
==> Linking _build/default/lib/folsom to _build/dev3/lib/folsom
==> Linking _build/default/lib/lager to _build/dev3/lib/lager
==> Linking _build/default/lib/recon to _build/dev3/lib/recon
==> Linking _build/default/lib/goldrush to _build/dev3/lib/goldrush
==> Compiling chp03
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
    c:/Users/JSTARNES/bootcamp/chp03/_build/dev3/lib
    c:/Users/JSTARNES/bootcamp/chp03/apps
    c:/Program Files/erl10.2/lib
==> Resolved chp03-0.1.0
==> release successfully created!

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp03
$

```

Figure 100 - make devrel command

19. Next, let's run the "make dev1-console;make dev2-console;make dev3-console" commands together on a single line as show in Figure 101.



```

MINGW64:/c:/Users/JSTARNES/bootcamp/chp03
jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp03
$ make dev1-console;make dev2-console;make dev3-console
"/c:/Users/JSTARNES/bootcamp/chp03/_build/dev1/rel/chp03/bin/chp03.cmd" console
"/c:/Users/JSTARNES/bootcamp/chp03/_build/dev2/rel/chp03/bin/chp03.cmd" console
"/c:/Users/JSTARNES/bootcamp/chp03/_build/dev3/rel/chp03/bin/chp03.cmd" console

jstarnes@JSTARNES529403 MINGW64 ~/bootcamp/chp03
$

```

Figure 101 – chp03 “make dev1-consoles;make dev2-console;make dev3-console” commands

20. Next, let's run the "application:which_applications()" function in each console window to see what are the current applications running, compare the 1st keyword value tuple of each list on from each window, your results should show that only the chp03_dev1 @<NodeName> is running the "{chp03,"An OTP application","0.1.0"}" application.
21. Next, let's run the observer application from any of the three console windows for chp03_dev1 or chp03_dev2 or chp03_dev3 and go to the Nodes menu and select chp03_dev1 @<NodeName> and the go to the Applications tab. You should notice that the chp03 application is listed for

chp03_dev. If you repeat this process by going to the Node menu and selecting the next two chp03 nodes available and go to their Applications tab, you will see that only chp03_dev1 is running the chp03 application as show in Figure 102.

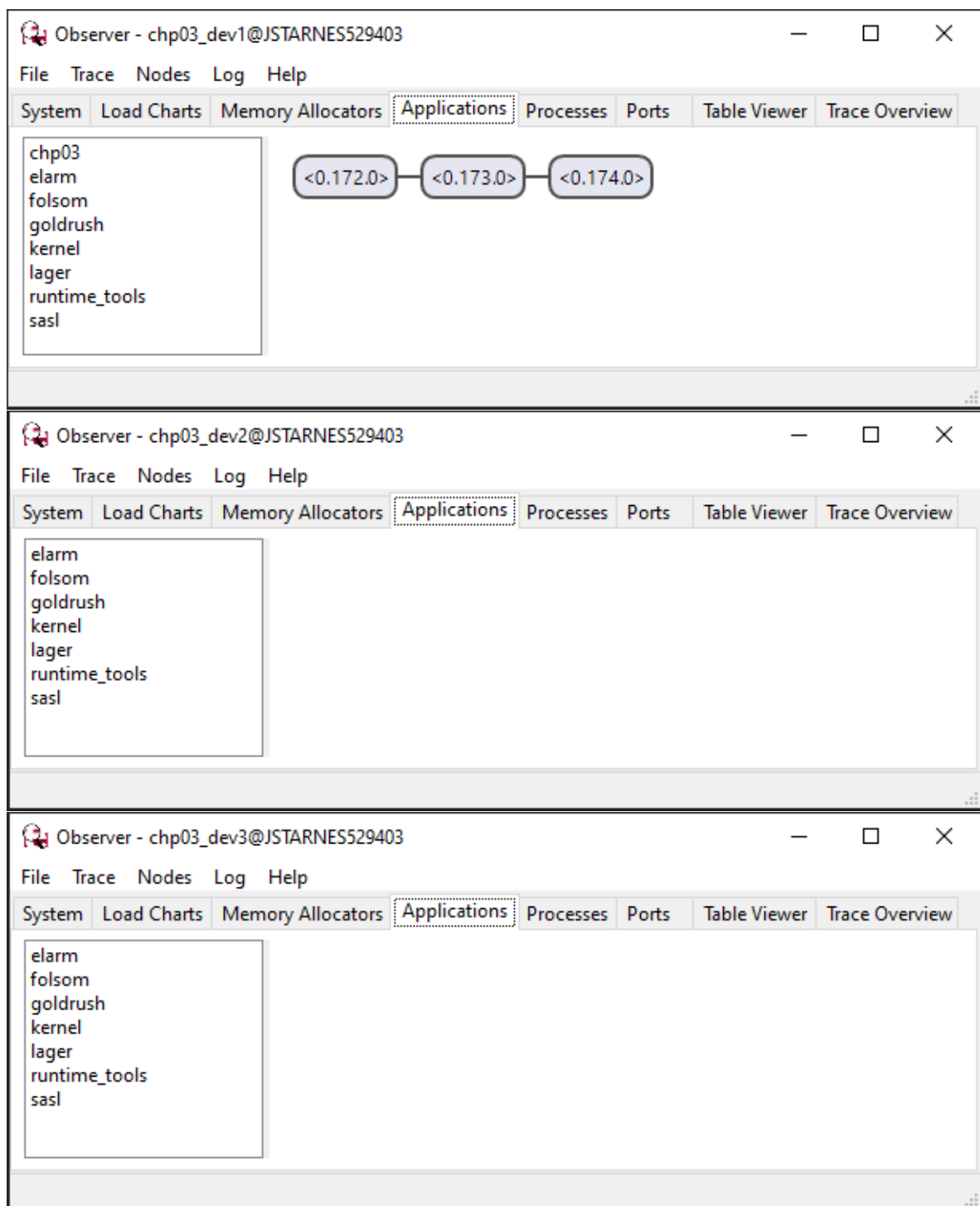
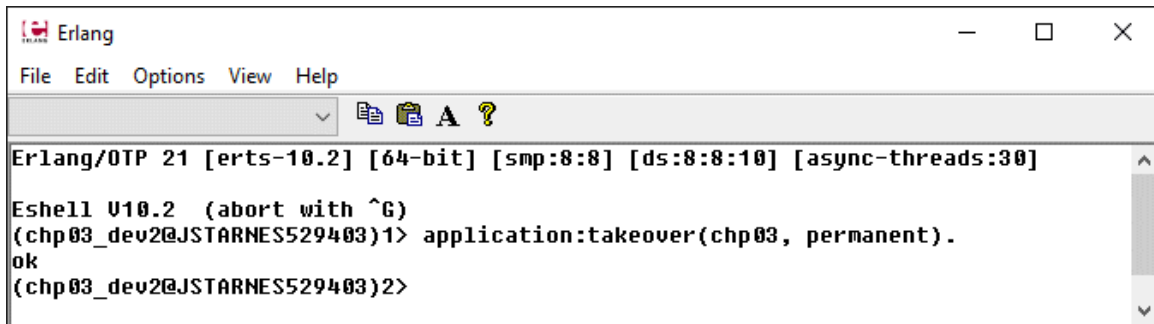


Figure 102 - Observer chp03_dev1, chp03_dev2, and chp03_dev3

22. Now, let's perform a takeover command from `chp03_dev2@<NodeName>` console by entering the `"application:takeover(chp03, permanent)."` function call and press return as shown in Figure 103. Figure 104 shows the result of a takeover on `chp03_dev1` after takeover on `chp03_dev2`. Figure 105 shows the result that `chp03_dev2` is now active after the takeover command.

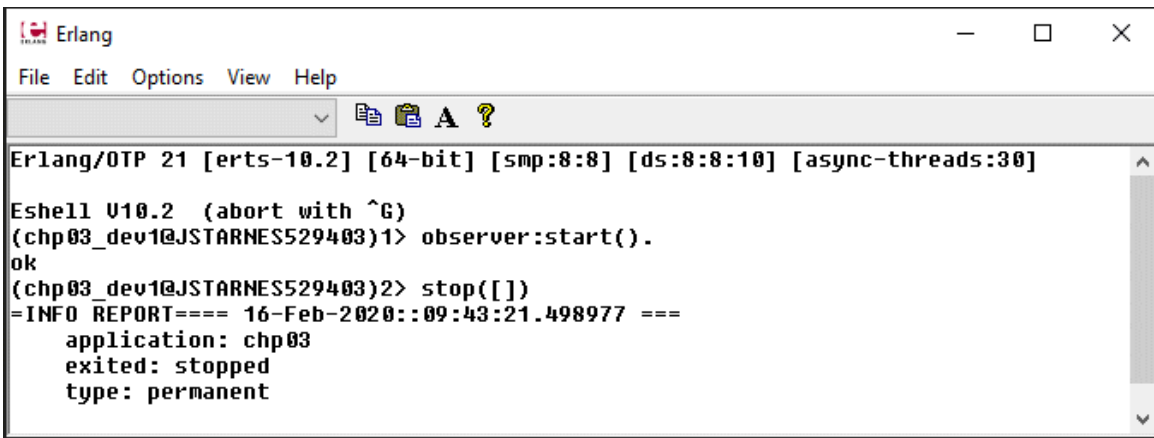


```

Erlang/OTP 21 [erts-10.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:30]

Eshell V10.2 (abort with ^G)
(chp03_dev2@JSTARNES529403)1> application:takeover(chp03, permanent).
ok
(chp03_dev2@JSTARNES529403)2>

```

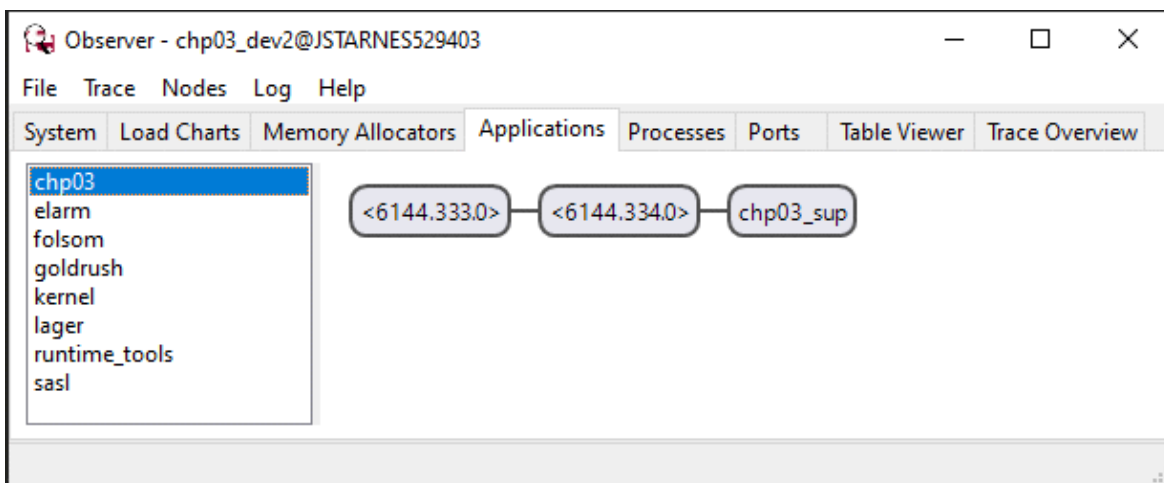
Figure 103 - `application:takeover(chp03, permanent)`


```

Erlang/OTP 21 [erts-10.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:30]

Eshell V10.2 (abort with ^G)
(chp03_dev1@JSTARNES529403)1> observer:start().
ok
(chp03_dev1@JSTARNES529403)2> stop([])
=INFO REPORT==== 16-Feb-2020::09:43:21.498977 ===
  application: chp03
  exited: stopped
  type: permanent

```

Figure 104 - `chp03_dev1` after takeover from `chp03_dev2`Figure 105 - `chp03_dev2` active after takeover

23. Next, let's simulate a node failure to induce an automatic failover instead of a manual takeover as did before, by entering `"init:stop([])"` as shown in Figure 106.

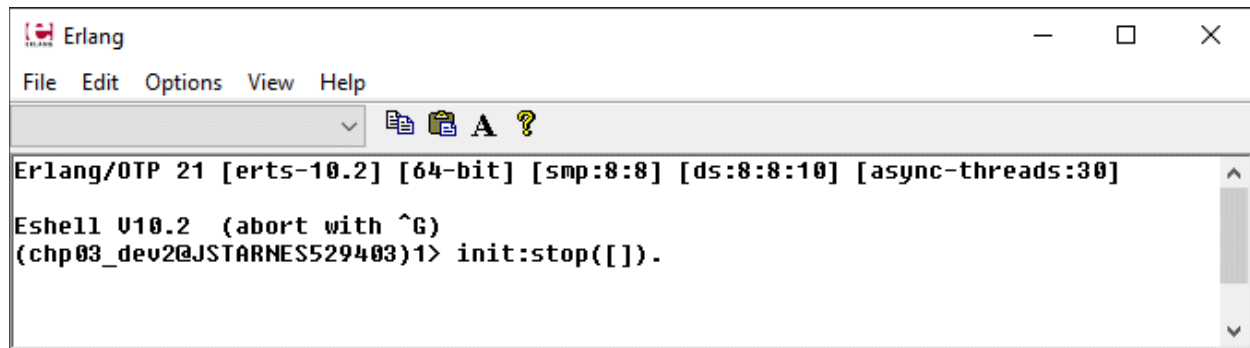


Figure 106 - chp03_dev2 simulate failover

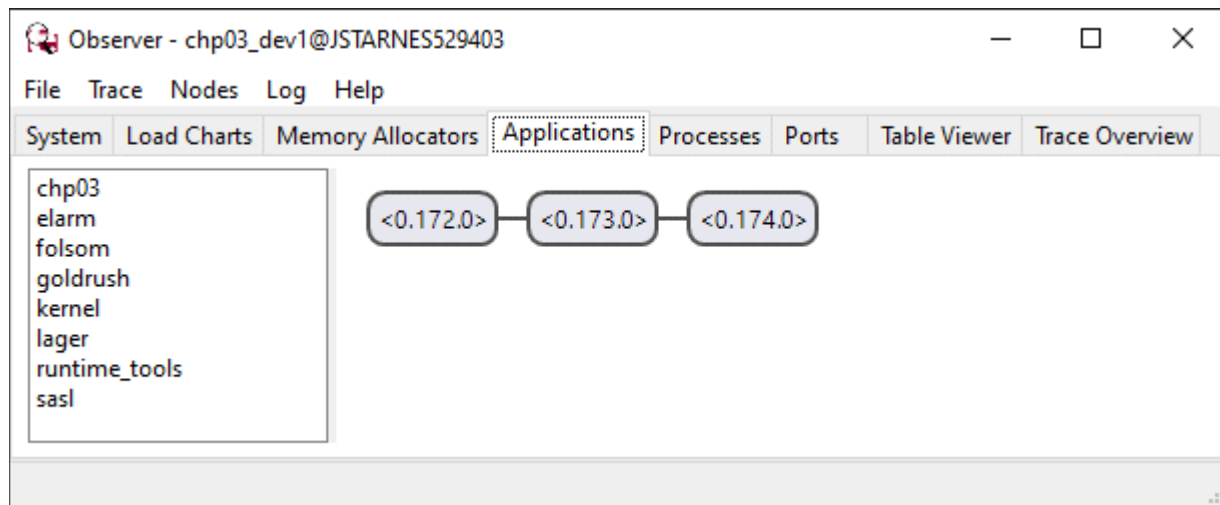


Figure 107 - chp03_dev1 active after failover on chp03_dev2

24. Next, let's view the sys.config file for chp03_dev1 in directory path “_build/dev1/rel/chp03/releases/0.1.0/sys.config” as shown in Listing 19. The distributed clause lists the applications chp03_dev1, chp03_dev2, and chp03_dev3 in order of priority.

Listing 19 - chp03_dev1 sys.config file

```
[
  {chp03, [1]},
  %% kernel
  {kernel, [
    {logger_sasl_compatible, true},
    {distributed, [{chp03, 5000, ['chp03_dev1@JSTARNES529403',
                                'chp03_dev2@JSTARNES529403',
                                'chp03_dev3@JSTARNES529403']}]},
    {sync_nodes_mandatory, ['chp03_dev1@JSTARNES529403',
                             'chp03_dev2@JSTARNES529403',
                             'chp03_dev3@JSTARNES529403']},
    {sync_nodes_timeout, 60000}
  ]},
  %% folsom config - [counter, gauge, histogram, history, meter, spiral, meter_reader]
  {folsom, [1]},
  %% SASL config
  {sasl, [
    {sasl_error_logger, {file, "dev1/log/sasl-error.log"}},
    {errlog_type, error},
    {error_logger_mf_dir, "dev1/log/sasl"}, % Log directory
    {error_logger_mf_maxbytes, 10485760}, % 10 MB max file size
    {error_logger_mf_maxfiles, 7} % 7 files max
  ]},
  %% Lager config - [debug, info, notice, warning, error, critical, alert, emergency, none]
  {lager, [
    {log_root, "dev1/log"},
    {crash_log, "crash.log"},
    {crash_log_msg_size, 65536},
    {crash_log_size, 10485760},
    {crash_log_date, "$D0"},
    {crash_log_count, 7},
    {handlers, [
      {lager_file_backend,
       [{file, "error.log"}, {level, error}, {size, 10485760}, {date, "$D0"}, {count, 50}]},
      {lager_file_backend,
       [{file, "console.log"}, {level, info}, {size, 10485760}, {date, "$D0"}, {count, 50}]},
      {lager_file_backend,
       [{file, "debug.log"}, {level, debug}, {size, 10485760}, {date, "$D0"}, {count, 50}]}
    ]}
  ]}
].
```

25. At this point, you may be asking yourself: Why do I have to waste all those resources on the standby nodes in a failover cluster? The answer is you don't, if you design your solution as a set of services, you start those services as top level applications, thus allowing for remote procedure calls (RPC) directly to those services. We already demonstrated this in Figure 98, as an rpc call to Erlang:node() function. However, now that we crossed the threshold of a distributed application, it is important to consider the network as an unreliable interface and plan accordingly to ensure that the rpc call succeeds reliably.

In Joe Armstrong's "Erlang a Language for Programming Reliable Systems" [6], he discusses the need to perform parallel requests across a set of nodes participating in a cluster to achieve higher availability. The discussion centers on the probability of a network failure during an RCP call, i.e., 10^{-3} or 0.1% percent, and that performing on four (4) nodes lowers the probability of a failure to 10^{-12} or 0.0000000001% percent. Future cluster topics such as strong vs weak consistency and consensus, and CAP theorem will be discussed in a future chapter.

26. A few more miscellaneous items to discuss, the `net_kernel` module contains functions related to distribution⁵. The `set_net_ticktime` and `get_net_ticktime` functions are for setting/getting the `net_ticktime`. The `monitor_nodes` is used for subscribing or unsubscribing for nodeup or nodedown notifications. There are also special case functions `net_kernel start/stop` for starting/stopping a node's participation in a cluster, the process remains after these calls. These are not to be confused with application start/stop functions, which stop the application.

Summary

We covered important topics for Erlang OTP Clusters and leveraged our basis template as a starting point for `basis_cluster` and `basis_cluster_failover` templates and we accomplished the following items:

- Update the templates in our `~/config/rebar3/templates` folder including new `basis`, `basis_cluster`, and `basis_cluster_failover` subfolders.
- Added GNU Make for Windows, so we can take advantage of make with rebar3.
- We discussed the latest trends of using virtual stacks such as VMware, Docker, or Kubernetes, etc. and compared to Erlang OTP Clusters, which are application based.
- We built our first cluster using the `basis_cluster`, which is a load-balancing type of cluster.
- We demonstrated how nodes can be joined by the `net_adm:ping` or `net_kernel:connect_node` functions.
- We explored usage of `erlang:node()` and `erlang:nodes()` functions to get a list of connected Erlang nodes.
- We discussed the `vm.args` config file contains the `-sname` or `-name` and the defined cookie for the cluster.
- We discussed usage of RPC calls to other nodes in the cluster
- We created a cluster that supports takeover and failover.
- We showed the cluster members of the `basis_cluster_failover` joining automatically at startup time and performed both takeover and failover scenarios.
- We used both `application:which_applications()` and `observer` to see which cluster member was active.
- We discussed the kernel distributed configuration and the priority order of the application
- We discussed the need for planning for network failure and usage of multiple rpc calls to guarantee a higher rate of success.

⁵ `net_kernel` - <https://erlang.org/doc/apps/kernel/application.html>

APPENDIX A ABBREVIATIONS AND ACRONYMS

The following acronyms and abbreviations may be found in this document:

APPENDIX B BIBLIOGRAPY

- [1] J. Armstrong, Programming Erlang, 2nd Edition - Software for a Concurrent World, The Pragmatic Bookshelf, 2013.
- [2] J. Amstrong, "A History of Erlang," June 2007. [Online]. Available: <https://dl.acm.org/doi/10.1145/1238844.1238850>. [Accessed 1 January 2020].
- [3] U. Wiger, "Four-fold Increase in Productivity and Quality," Ericsson Telecom AB, 30 01 2001. [Online]. Available: <https://pdfs.semanticscholar.org/dd95/9d83ea1052da4fdf6d197f046a3c3013c4f7.pdf>. [Accessed 1 January 2020].
- [4] J. L. Andersen, "ERLANG IN PRODUCTION," Erlang Solutionms, 11 June 2014. [Online]. Available: <https://www.erlang-factory.com/static/upload/media/1403620025208515erlanginproduction.pdf>.
- [5] B. Benavides, "The Extinction of the Dodos (OTP Style)," 1 11 2016. [Online]. Available: <https://medium.com/erlang-battleground/the-extinction-of-the-dodos-otp-style-f421f9de4275>.
- [6] J. Armstrong, "QCon London," 13 March 2009. [Online]. Available: https://qconlondon.com/london-2009/qconlondon.com/dl/qcon-london-2009/slides/JoeArmstrong_ErlangALanguageForProgrammingReliableSystems.pdf.
- [7] Basho, "Basho/lager," Basho, [Online]. Available: <http://github.com/basho/lager#user-content-internal-log-rotation>.