Week 6 Module

During this week we will be looking at logging and monitoring.   When our applications run, we really have no idea what they are doing under the covers.   All we have is the GUI presented to the user but there are many times when we need to know what the system is doing.   This is logging and vital to any production level application.

Objectives

During this week, you will:

* Demonstrate an understanding of Logging
* Create notifications based on events
* Hook an application to an ELK server for logging, monitoring, and reporting

Readings

Each week I have provided a number of *Enrichment Readings* which I have identified to provide additional information either more generally or more specifically as needed. These readings are recommended if you find you have unanswered questions. You may also use the course discussions to ask lingering questions or follow-up on a thought.

There are also **Required Readings** which you will need to complete in order to be successful in the course. These are clearly identified in the module but also listed below.

* [*Bitnami ELK Stack for AWS Cloud*  (Links to an external site.)](https://docs.bitnami.com/aws/apps/elk/)from docs.bitnami.com
* [*Exceptional logging of exceptions in Python is Cloud Computing?* (Links to an external site.)](https://www.loggly.com/blog/exceptional-logging-of-exceptions-in-python/) from Aaron Maxwell at loggly.com

# logging of exceptions in Python

## The “Big Tarp” Pattern

try:

main\_loop()

except Exception:

logger.exception("Fatal error in main loop")

This is a broad catch-all. It is suitable for some code path where you know the block of code (i.e, main\_loop()) can raise a number of exceptions you may not anticipate. And rather than allow the program to terminate, you decide it’s preferable to log the error information, and continue from there.

Note that you don’t have to pass the exception object here. You do pass a message string. This will log the full stack trace, but prepend a line with your message.

notice that the first line is the message you passed to logger.exception(), and the subsequent lines are the full stack trace, including the exception type (ZeroDivisionError in this case). It will catch and log any kind of error in this way.

By default, logger.exception uses the log level of ERROR. Alternatively, you can use the regular logging methods— logger.debug(), [**logger.info**](http://logger.info/)(), logger.warn(), etc.—and pass the exc\_info parameter, setting it to True:

while True:

try:

main\_loop()

except Exception:

logger.error("Fatal error in main loop", exc\_info=True)

Setting exc\_info to True will cause the logging to include the full stack trace…. exactly like logger.exception() does. The only difference is that you can easily change the log level to something other than error: Just replace logger.error with logger.warn, for example.

**The “Pinpoint” Pattern**

from openburrito import find\_burrito\_joints, BurritoCriteriaConflict

# "criteria" is an object defining the kind of burritos you want.

try:

places = find\_burrito\_joints(criteria)

except BurritoCriteriaConflict as err:

logger.warn("Cannot resolve conflicting burrito criteria: {}".format(err.message))

places = list()

The pattern here is to optimistically execute some code—the call to find\_burrito\_joints(), in this case—within a try block. In the event a specific exception type is raised, you log a message, deal with the situation, and move on.

The key difference is the except clause. With the Big Tarp, you’re basically catching and logging any possible exception. With Pinpoint, you are catching a very specific exception type, which has semantic relevance at that particular place in the code.

**The “Transformer” Pattern**

try:

something()

except SomeError as err:

logger.warn("...")

raise DifferentError() from err

Here, you are catching an exception, logging it, then raising a different exception.

You will want to use this pattern when an exception may be raised that does not map well to the logic of your application. This often occurs around library boundaries.

In Python 3, exceptions can now be *chained*. The raise ... from ... syntax provides this. When you say raise NoMatchingRestaurants(criteria) from err, that raises an exception of typeNoMatchingRestaurants. This raised exception has an attribute named \_\_cause\_\_, whose value is the instigating exception. Python 3 makes use of this internally when reporting the error information.

How do you do this in Python 2? Well, you can’t.

**The “Message and Raise” Pattern**

In this pattern, you log that an exception occurs at a particular point, but then allow it to propagate and be handled at a higher level:

try:

something()

except SomeError:

logger.warn("...")

raise

**The “Cryptic Message” Antipattern**

**The Most Diabolical Python Antipattern**

try:

something()

except Exception:

pass

# **Why Logging**

Logging is one of the most crucial aspects of software engineering.  During the execution of your application, you need to know what the application is doing, where it is, and get good error information.

Logging has come a long way since the early days of software engineering.   Back when we used C exclusively to do hardcore development, the only logging we had available was printf.    At various places in the code, we would place printf("Here"); statements.   During the execution of the application, we were able to determine where in the flow the application was currently executing.

Of course, this type of logging was extremely tedious.   As tools advanced, we were given the ability to debug our applications using a standalone debugger.   We didn't have IDEs yet.  There was no visual studio nor eclipse available.   You ran your application and attached a debugger to it.   Once the IDE became available, we had the power to debug and step through our code.

In all of this though, we really just used logging as a way to debug our code.   We weren't monitoring the application persay.

When newer languages became available like C# and Java, the world of logging really opened up.   Log4J is one class library for Java and I might consider to be the catalyst into advanced logging.   We now had a logging toolkit that would handle all of the details for us.   Log file sizes, rotation, levels, etc.

The final piece in the logging/monitoring puzzle is application log drains....ELK for example.   Systems were we can ship our logs, setup visualization and monitor what our applications are doing.   All of the pieces are in place now to give you complete control over your applications and what they are doing.

Remote logging (DENver startup-week)

What to log:

User input (any user input)

System broadcasts

Authorizations/signins

API requests (not streams)

Heavy operations (compute and IO)

What to NOT log:

Log in loops

Log redundantly

Log noise

# **What is ELK**

ELK stands for Elasticsearch, Logstash, and Kibana.   Technically ELK isn't an application but a stack of application.   It is designed to give us a picture of our application based on log messages that we provide to the ELK stack through our application

The idea is to centralize logs for troubleshooting purposes. If we “report” all applications to the same central location. We only have to look one place to find information on bugs, etc.

Collect data

Clean data

Convert into structured form

Analyze data

We need this for

Issues (general bugs)

Security issues

Internet of things (remote access thermostats, fridges, etc.)

Predictive analysis

Performance analysis

ELK is a combination of 3 open source separate products from separate vendors

Elasticsearch

Stores logs in JSON format and allow search in data

LogStash

Central data pipeline tool that can collect and parse structured / unstructured data and events

Kibana

Web interface that allow search, display and summarization, charting

Permits saving and sharing of logs/snapshots

How do ELK work

Logs are collected in a central place and indexed and we can create visual representations

# **Adding Logging to Python**

def add(x, y):

return x + y

def substract(x, y):

return x – y

def multiply(x, y):

return x \* y

def divide(x, y):

return x / y

num1 = 10

num2 = 5

addResult = add(num1, num2)

#print(‘Add: {} + {} = {}’.format(num1, num2, addResult)

subResult = add(num1, num2)

#print(‘Add: {} - {} = {}’.format(num1, num2, subResult)

mulResult = add(num1, num2)

#print(‘Add: {} \* {} = {}’.format(num1, num2, mulResult)

divResult = add(num1, num2)

#print(‘Add: {} / {} = {}’.format(num1, num2, divResult)

Logging

import logging

#logging levels (from Python documentation)

#DEBUG: detailed information, typically of interest only when diagnosing problems.

#INFO: Confirmation that things are working as expected.

#WARNING (default level): An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.

#ERROR: Due to a more serious problem, the software has not been able to perform some function.

#CRITICAL: A serious error, indicating that the program itself may be unable to continue running.

Instead of ‘print’ we could use ‘logging.debug’ – logs to console

Default it will log only warnings and higher – so above wouldn’t log anything

Changing it to ‘logging.warning’ will print i.e.

WARNING:root:Add: 10 + 5 = 15

Change default logging level (+ add file + add formatting):

logging.basicConfig(filename=’test.log’, level=logging.DEBUG, format=’%(asctime)s:%(levelname)s:%(message)s’)

by default, the info in file will be appended when program is rerun

printout would be:

2019-03-10 15:52:13,118:DEBUG:Add: 10 + 5 = 15

All formatting options: Python documentation 16.6.7 LogRecord Attributes

import logging

logging.basicConfig(filename=’filename’, level=logging.INFO, format=’ :%(levelname)s:%(message)s’’))

different loggers can be formatted differently

if we create and configure a root logger in a unit that we import into our main unit and here also create and configure a root logger, the main logger will be ignored because a root logger is already configured (global code is run as soon as the code is included, in the import statement). If the loggers are logging at the same level, the main unit’s logs will be inserted into the unit’s logfile.

We can however create different loggers

logger = logging.getLogger(\_\_name\_\_) #\_\_name\_\_ is sorta convention – will make a log file with the unit name

since we created a new logger variable, we should be logging to that instead of logging. So, our code should be changed to:

logger.info(‘Create Employee: bla bla’)

these loggers are in a stack – if our new logger doesn’t have something set, it will grab that setting from the root logger

to get this into a new file, we need to create a new file handler:

logger = logging.getLogger(\_\_name\_\_)

logger.setLevel(logging.INFO)

formatter = logging.Formatter(’%(asctime)s:%(levelname)s:%(message)s’)

file\_handler = logging.filehandler(‘logfilename’))

file\_handler.setFormatter(formatter)

logger.addHandler(file\_handler)

so now we don’t need the basic configuration anymore…

file handlers can have a log level set, rather than setting the level in the logger

file\_handler.setKLevel(logging.ERROR)

def divide(x, y):

try:

result = x / y

except ZeroDivisionError:

logger.exception(‘Division by zero’)

else:

return result

if we log an exception instead of an “error” we’ll get the tracestack in the log file “for free”

we can create a stream handler (instead of a file handler) to stream to console window

stream\_handler = logging.StreamHandler()

stream\_handler.setFormatter(formatter)

logger.addHandler(stream\_handler)

# **ELK Configuration Changes**

In this section, we will start the process of using the ELK AWS instance you set up in week 5.  We will walk through the process of making a few configuration changes in ELK and the process of adding logging to a python application which pushes the logs to ELK for further processing and visualization.

There are a few changes we need to make to the ELK instance you setup in week 5.   Currently, the ELK instance you have setup is configured to process the access\_log from apache on the machine where ELK is installed.   This works great for validating the system is running but really mucks up trying to see our own logging so we will make a few changes to the configuration.

Here are the basic steps that you need to perform:

1) Convert your PEM key file to a ppk format for use in Putty and import into Putty.  
Sorry this is for Windows...  
[https://docs.bitnami.com/aws/faq/starting-bitnami-aws/connect\_ssh/ (Links to an external site.)](https://docs.bitnami.com/aws/faq/starting-bitnami-aws/connect_ssh/)

2) Launched Putty to access the ELK instance.   Notice that if you use the keys and the username of bitnami, you don't need a password.    You will be dumped right into the command line of the Linux OS.

3) Navigate to /opt/bitnami

4) Stop logstash with the following:  
sudo ./ctlscript.sh stop logstash

5) Navigate to /opt/bitnami/logstash/modules  
remove the two folders in this directory: fb\_apache and netflow

6) Naviate to /opt/bitnami/logstash/conf  
Change the logstash.conf to read like:

input  
{  
  tcp  
  {  
    mode => "server"  
    host => "0.0.0.0"  
    port => 5010  
  }  
  udp  
  {  
    host => "0.0.0.0"  
    port => 5959  
    codec => json  
  }  
}

output  
{

  elasticsearch  
  {  
    hosts => ["127.0.0.1:9200"]

    index => "python-application-log"  
  }

}

7) Go back to /opt/bitnami and start logstash

    sudo ./ctlscript.sh start logstash

You are now no longer monitoring the apache access log and just waiting on TCP or UDP messages to arrive.

Setting up an Index

Logstash is the first application in the ELK stack.  It receives logs and stores the data in Elasticsearch.   In order to Kibana to make sense of the stored data, we use indexes.   You will notice in the previous section, we told logstash to accept data on TCP and UDP and store in elasticsearch using an index called python-application-log.   Logstash can be configured to store data using a variety of indexes but for our purposes we are just using the one.

Logstash doesn't really know anything about the index.   It is just the name of a container the data is assigned.  Elasticsearch is the part of the stack that needs to know about the index - it will be receiving data from logstash and told to use a specific index.   Elasticsearch would like to know about that index before hand and that's what the code below will do.

You will need to launch Putty again and get to a command line interface.   Once there, copy and paste the following code into the command line.   This code will run curl. Curl is just an application that can transmit data using a variety of protocols.  In this instance, we will be telling curl to access the Elasticsearch web service located at port 9200 on the ELK server and send a JSON message with the configuration we will be using.

curl -XPUT http://localhost:9200/python-application-log -H "content-type: application/json" -d @- <<EOF  
{  
"mappings": {  
"dynamic": false,  
"properties": {  
"level": { "type": "keyword" },  
"test\_boolean": { "type": "keyword" },  
"test\_list": { "type": "text" },  
"test\_integer": { "type": "text" },  
"test\_float": { "type": "text" },  
"host": { "type": "keyword" },  
"@timestamp": { "type": "date" },  
"@message": { "type": "text" },  
"message": { "type": "text" }  
}  
}  
}  
EOF

When this index is put in place, elasticsearch will take the JSON that appears on the TCP/UDP port, send by logstash, and parse the key values presented into what the index pattern says.   Any fields NOT in the index will simple be text fields.   Notice the dynamic:false.   This flag tells elasticsearch that if additional fields are found, don't error but just accept the new fields as text.

One of the keys to the index is the keyword "type".   The keyword "type" tells elasticsearch and more importantly kibana that the key associated can be used in a search.   Notice we have the 'level' key as a keyword.  This will be important to us shortly.

When building indexes, if something goes wrong, you can delete the index using the code:

* curl -X DELETE "localhost:9200/python-application-log"

# **Python Logging**

Okay, now at this point, we have ELK setup and listening for JSON messages on either TCP or UDP.  Our next step is configure a python application to use the ELK stack for our logging 'drain'.

For this demonstration, we used logstash-python as the module at handle the communication between the built-in logging functionality of python and the ELK stack.   Note, I'm using Python 2.7 in the video.   Based on experimentation, Python 3 broke some of the functionality in log-stash python module.

There are really three main aspects to adding the ELK functionality to the python application.

1. **Importing** **Logstash**.   This gives us access to the LogstashHandler that takes care of sending the log information to ELK.
2. **Configuring Logstash Access**.  This starts with adding the handler to our logging object using the statement  
     
   .addHandler(logstash.LogstashHandler(<IP>, 5959, version=1))  
     
   Note port 5959 was specified in the logstash.conf file we changed in the previous section.   You will need to put the IP address or domain name of your ELK stack in place of <IP>.   You can find that using either ipconfig or on the AWS console.  
     
   You might also recall the security of our ELK server running on AWS...What we are saying in the code above is when logging, our application should contact our ELK server on port 5959 and transmit error information through logstash.   Based on the security setup of ELK from week 5, does it currently allow access to port 5959?   If not, we might have to update the security in AWS to allow access to this specific port against our ELK server IP

1. **Calling the right logging level method and providing the necessary information.**   The entire JSON object sent will be mapped to message in Elasticsearch.  We can provide additional 'extra' parameter as shown in the example application.

import logging

import logstash

import sys

import time

test\_logyger = logging.getLogger(‘Python-logstash-logger’)

test\_logger.setLevel(logging.INFO)

test\_logger.addHandler(logstash.LogstashHandler(’34.233.71.31’, 5959, version-1))

#IP of ELK stack – port# (UDP5959),

test\_logger(error(‘Python logstash: test logstash error message.’)

test\_logger.info(‘Python-logstash: test logstash infor message.’)

test\_logger.warning(python-logstatsh: test logstash warnng message.’)

extra = {

‘test\_string’: ‘python version: ‘ + repr(sys.version\_info),

‘test\_boolean’: True,

‘test\_dict’: {‘a’: 1, ‘b’: ‘c’},

‘test\_float’: 1.23,

‘test\_integer’: 123,

‘test\_list’: [1, 2, 3],

}

test\_logger.info(python-logstash: test extra fields, extra=extra)

install python-logstash(0.4.6) for python 2.7

# **Setting up Kibana**

The K in the ELK stack is Kibana.   Kibana is a visualization tool that allows you to 'see' the data in elasticsearch.   It is easy to use and very powerful.   There is some setup we need to do touched on in the last section.

In the video, we:

1) Created an index pattern based on the index we created for elasticsearch

2) Created visualizations using the keywords defined in our index.

3) Created a dashboard using the visualizations.

# **Real World ELK Use**

Indexing 200 million

Needs:

High scalability, filtering, querying, low latency and relyability

Elastic search is a search engine allowing real time and highly available

Loading is a 5-step process

Fetch user even data and launch instances

De-duplicate daily records (EMR)

Index into cluster (Java Load Process)

Optimize cluster (ES API)

Snapshot to S3 (backup)

a.k.a.

pre-run activity

EMR activity

Report activity

Post-run activity

Backup activity

Tracking what’s happening on a network and giving a security analyst a place to look in an attempt to discover what users are doing what

/opt/logstash/bin/logstash -e 'input { stdin { } } output { elasticsearch { hosts => ["localhost "] } }' --path.data /root/data