

# Yearbook 2021-22

Opensource.com Correspondent Program

# We are Opensource.com

Opensource.com is a community website publishing stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Do you have an open source story to tell? Submit a story idea at [opensource.com/story](https://opensource.com/story)

Email us at [open@opensource.com](mailto:open@opensource.com)



Supported by  
**Red Hat**

## Table of Contents

Convert your Raspberry Pi into a trading bot with Pythonic.....	3
Manage Linux users' home directories with systemd-homed.....	23
An open source developer's guide to systems programming.....	27
What I miss about open source conferences.....	38
What you need to know about cluster logging in Kubernetes.....	40
Create an app with this Arnold Schwarzenegger-themed programming language.....	49
Monitor your home's temperature and humidity with Raspberry Pi and Prometheus.....	54
How to update a Linux symlink.....	64
5 common bugs in C programming and how to fix them.....	67
Calculate date and time ranges in Groovy.....	76
How a college student founded a free and open source operating system.....	81
7 summer book recommendations from open source enthusiasts.....	87
A programmer's guide to GNU C Compiler.....	92
Analyze the Linux kernel with ftrace.....	98
5 ways to involve people who don't write code in the DevOps process.....	108
9 open source alternatives to try in 2022.....	112
Share your Linux terminal with tmate.....	116
What's new with Java 17 and containers?.....	120
A simple CSS trick for dark mode.....	123
How I automate plant care using Raspberry Pi and open source tools.....	126
5 Rust tools worth trying on the Linux command line.....	129
4 Linux tools to erase your data.....	133
5 agile mistakes I've made and how to solve them.....	136
Run containers on Mac with Lima.....	141

# Convert your Raspberry Pi into a trading bot with Pythonic

By Stephan Avenwedde

The current popularity of cryptocurrencies also includes trading in them. Last year, I wrote an article [How to automate your cryptocurrency trades with Python](#) which covered the setup of a trading bot based on the graphical programming framework [Pythonic](#), which I developed in my leisure. At that time, you still needed a desktop system based on x86 to run Pythonic. In the meantime, I have reconsidered the concept (web-based GUI). Today, it is possible to run Pythonic on a Raspberry Pi, which mainly benefits the power consumption because such a trading bot has to be constantly switched on.

That previous article is still valid. If you want to create a trading bot based on the old version of Pythonic (0.x), you can install it with `pip3 install Pythonic==0.19`.

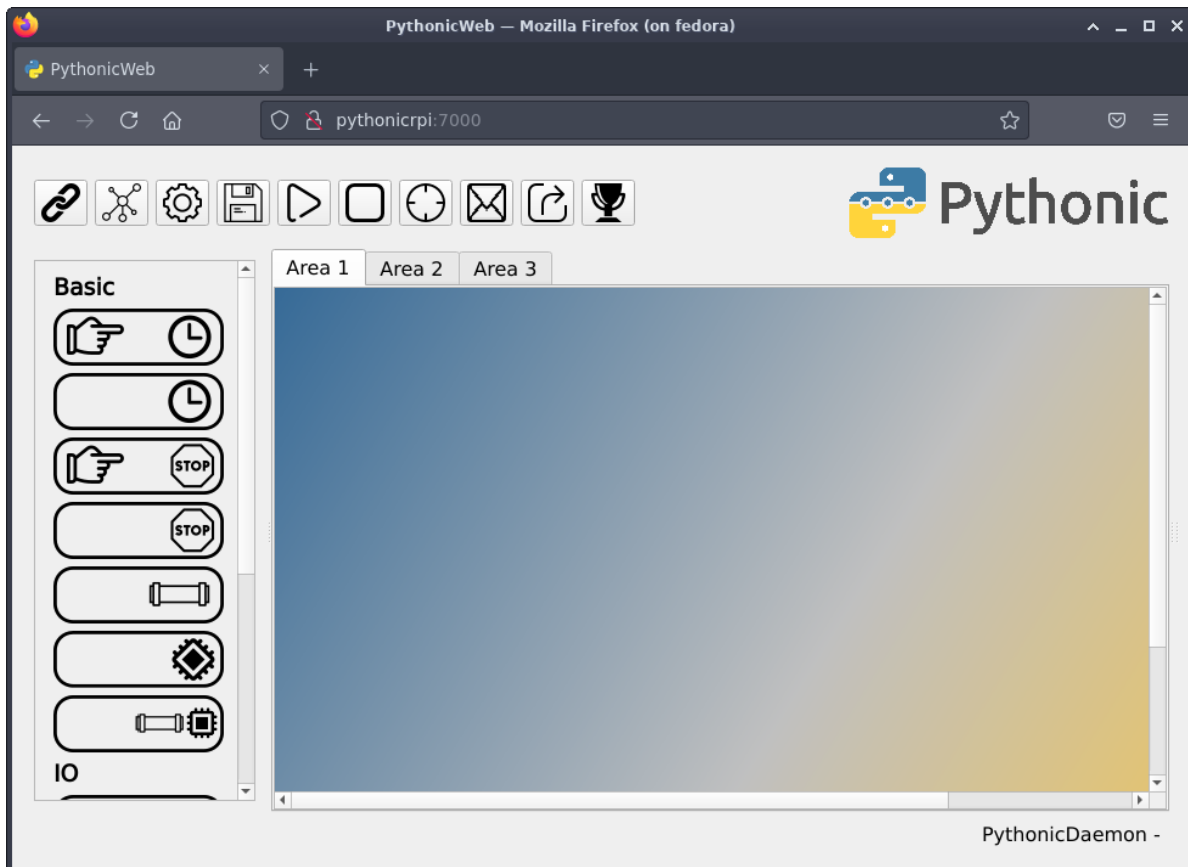
This article covers the setup of a trading bot running on a Raspberry Pi and executing a trading algorithm based on the [EMA crossover strategy](#).

## Install Pythonic on your Raspberry Pi

Here, I only briefly touch on the subject of installation because you can find detailed installation instructions for Pythonic in my last article [Control your Raspberry Pi remotely with your smartphone](#). In a nutshell: Download the Raspberry Pi image from [sourceforge.net](#) and flash it on the SD card.

The PythonicRPI image has no preinstalled graphical desktop, so to proceed, you should be able to access the programming web GUI (<http://PythonicRPI:7000/>):





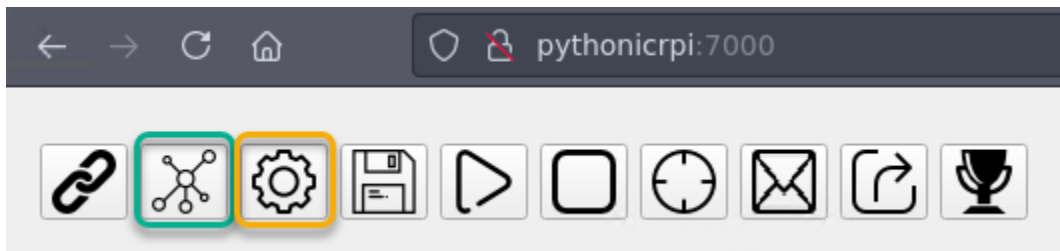
(Stephan Avenwedde, [CC BY-SA 4.0](#))

## Example code

Download the example code for the trading bot from [GitHub](#) (direct download link) and unzip the archive. The archive contains three different file types:

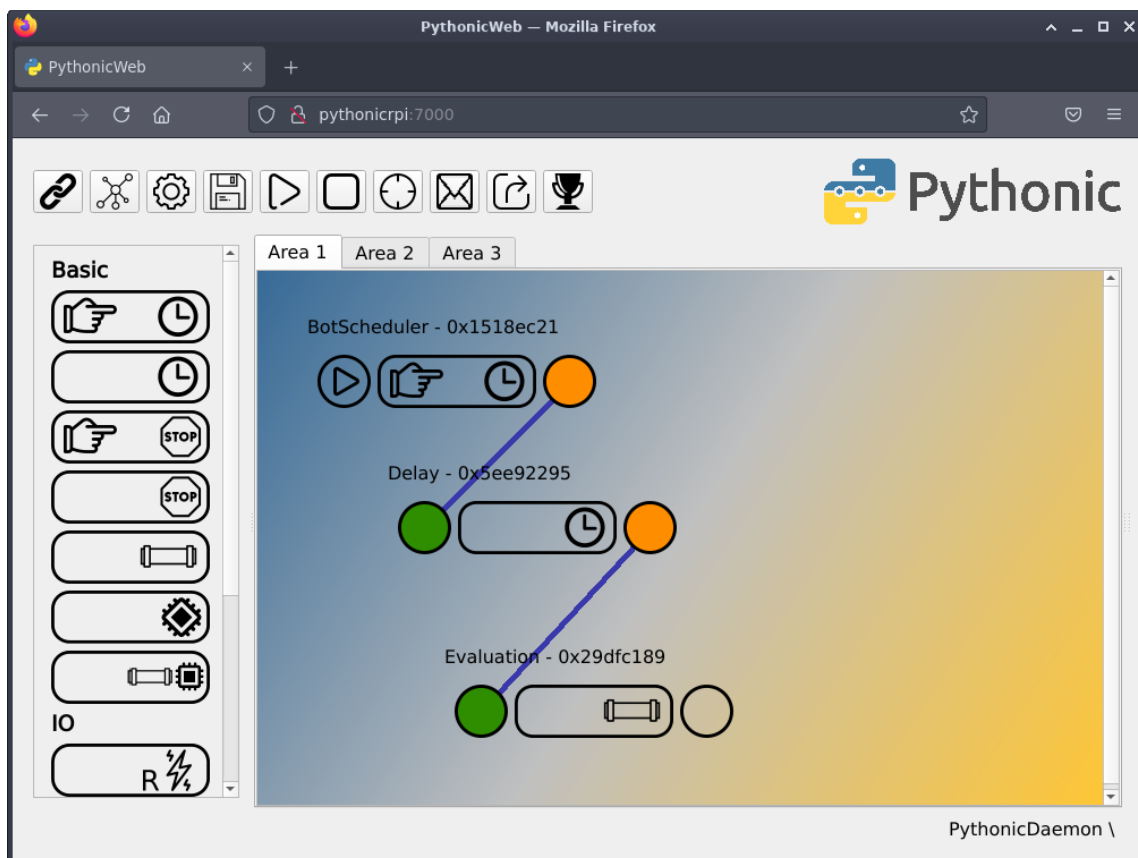
- `\*.py`-files: Contains the actual implementation of certain functionality
- `current_config.json`: This file describes the configured elements, the links between the elements, and the variable configuration of elements
- `jupyter/backtest.ipynb`: A [Jupyter](#) notebook for backtesting
- `jupyter/ADAUSD_5m.df`: A minimal OHLCV dataset which I use in this example

With the green outlined button, upload the `current_config.json` to the Raspberry Pi. You can upload only valid configuration files. With the yellow outlined button, upload all the `\*.py` files.



(Stephan Avenwedde, [CC BY-SA 4.0](#))

The `\*.py` files are uploaded to `/home/pythonic/Pythonic/executables` whereas the `current_config.json` is uploaded to `/home/pythonic/Pythonic/current_config.json`. After uploading the `current_config.json`, you should see a screen like this:

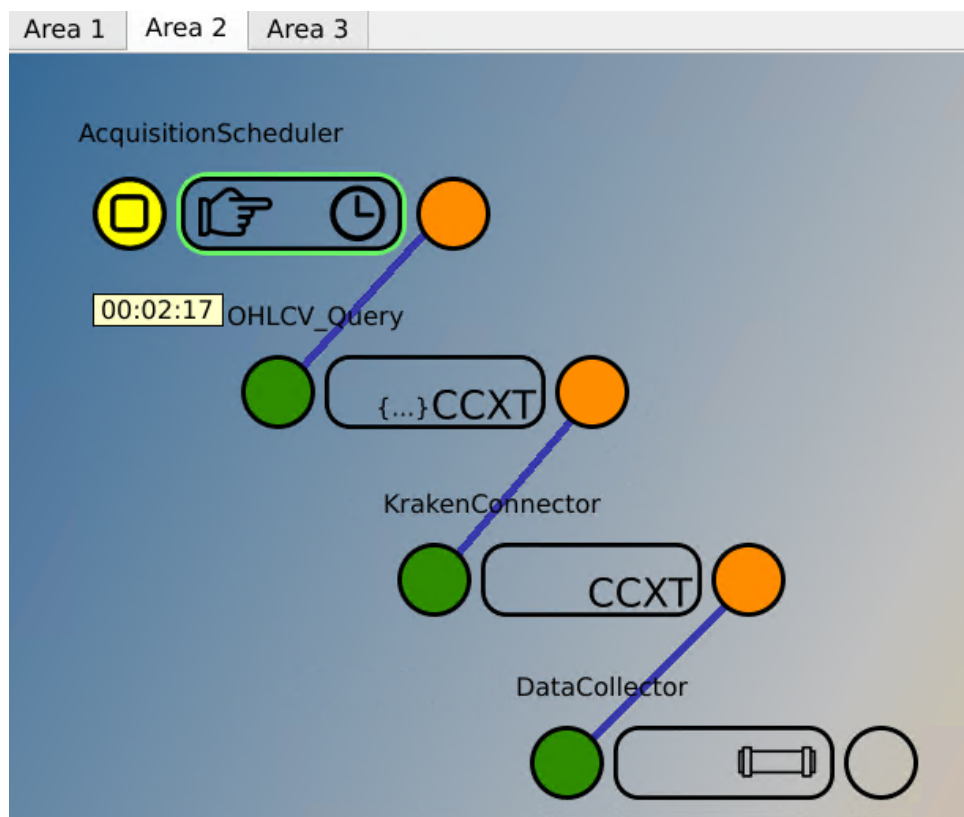


(Stephan Avenwedde, [CC BY-SA 4.0](#))

Now I'll go step-by-step through each part of the trading bot.

## Data acquisition

Like in the last article, I begin with the data acquisition:



(Stephan Avenwedde, [CC BY-SA 4.0](#))

The data acquisition can be found on the **Area 2** tab and runs independently from the rest of the bot. It implements the following functionality:

- **AcquisitionScheduler**: Trigger subsequent elements every five minutes
- **OHLCV\_Query**: Prepares the OHLCV query method
- **KrakenConnector**: Establishes a connection with the Kraken cryptocurrency exchange
- **DataCollector**: Collect and process the new OHLCV data

The *DataCollector* gets a Python list of OHLCV data with a prefixed timestamp and converts it into a [Pandas DataFrame](#). Pandas is a popular library for data analysis and manipulation. A *DataFrame* is the base type for data of any kind to which arithmetic operation can be applied.

The task of the DataCollector (`generic_pipe_3e059017.py`) is to load an existing DataFrame from file, append the latest OHLCV data, and save it back to file.

```
import time, queue
import pandas as pd
from pathlib import Path
```

```

try:
    from element_types import Record, Function, ProcCMD, GuiCMD
except ImportError:
    from Pythonic.element_types import Record, Function, ProcCMD, GuiCMD
class Element(Function):
    def __init__(self, id, config, inputData, return_queue, cmd_queue):
        super().__init__(id, config, inputData, return_queue, cmd_queue)

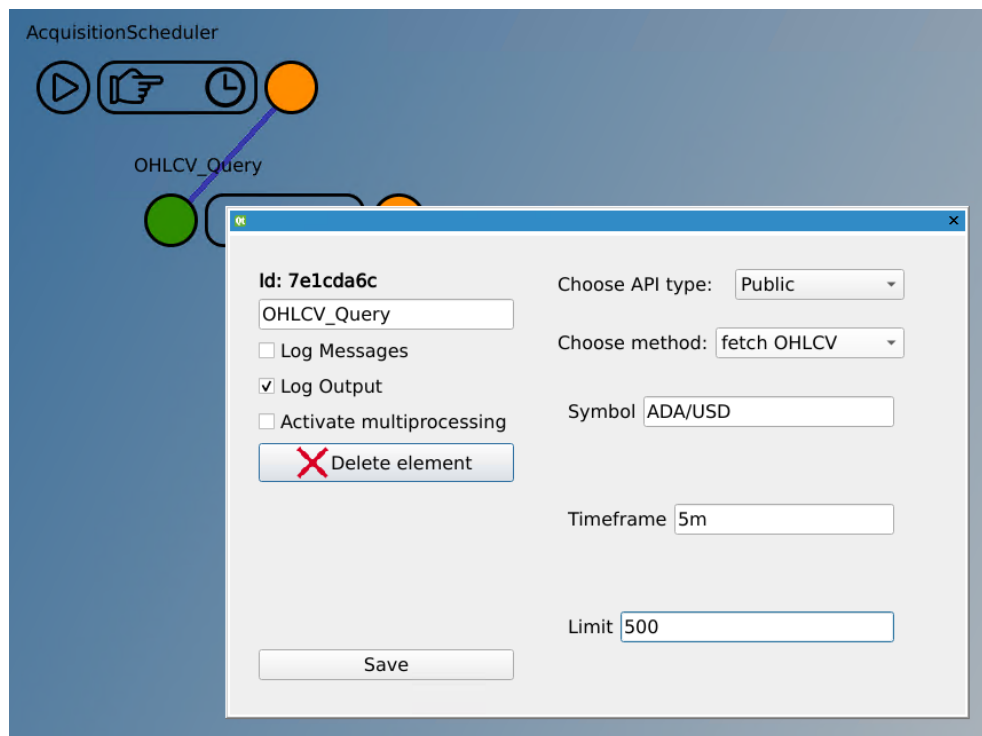
    def execute(self):
        df_in = pd.DataFrame(self.inputData, columns=['close_time', 'open',
'high', 'low', 'close', 'volume'])
        df_in['close_time'] = df_in['close_time'].floordiv(1000) # remove
milliseconds from timestamp
        file_path = Path.home() / 'Pythonic' / 'executables' / 'ADAUSD_5m.df'
        try:
            # load existing dataframe
            df = pd.read_pickle(file_path)
            # count existing rows
            n_row_cnt = df.shape[0]
            # concat latest OHLCV data
            df = pd.concat([df, df_in],
ignore_index=True).drop_duplicates(['close_time'])
            # reset the index
            df.reset_index(drop=True, inplace=True)
            # calculate number of new rows
            n_new_rows = df.shape[0] - n_row_cnt
            log_txt = '{}: {} new rows written'.format(file_path, n_new_rows)

        except Exception as e:
            log_txt = 'File error - writing new one'
            df = df_in
        # save dataframe to file
        df.to_pickle(file_path)
        logInfo = Record(None, log_txt)
        self.return_queue.put(logInfo)

```

This code is executed every full five minutes as the OHLCV data is also in 5-minute intervals.

By default, the *OHLCV\_Query* element only downloads the dataset for the latest period. To have some data for developing the trading algorithm, right-click the **OHLCV\_Query** element to open the configuration, set the *Limit* to 500, and trigger the **AcquisitionScheduler**. This causes the download of 500 OHLCV values:



(Stephan Avenwedde, [CC BY-SA 4.0](#))

## Trading strategy

Our trading strategy will be the popular [EMA crossover strategy](#). The EMA indicator is a weighted moving average over the last  $n$  close prices that gives more weight to recent price data. You calculate two EMA series, one for a longer period (for example,  $n = 21$ , blue line) and one for a shorter period (for example,  $n = 10$ , yellow line).



(Stephan Avenwedde, [CC BY-SA 4.0](#))

The bot should place a buy order (green circle) when the shorter-term EMA crosses above the longer-term EMA. The bot should place a sell order when the shorter-term EMA crosses below the longer-term EMA (orange circle).

## Backtesting with Jupyter

The example code on [GitHub](#) (direct download link) also contains a [Jupyter Notebook](#) file (`backtesting.ipynb`) which you use to test and develop the trading algorithm.

**Note:** Jupyter is not preinstalled on the Pythonic Raspberry Pi image. You can either install it also on the Raspberry Pi or install it on your regular PC. I recommend the latter, as you will do some number crunching that is much faster on an ordinary x86 CPU.

Start Jupyter and open the notebook. Make sure to have a DataFrame, downloaded by the *DataCollector*, available. With **Shift+Enter**, you can execute each cell individually. After executing the first three cells, you should get an output like this:

```
In [9]: import pandas as pd
        from pathlib import Path
```

```
In [10]: file_path = Path.home() / 'Pythonic' / 'executables' / 'ADAUSD_5m.df'
         ohlcv = pd.read_pickle(file_path)
```

```
In [11]: ohlcv
```

Out[11]:

	close_time	open	high	low	close	volume
0	1629297600	2.005271	2.007492	2.005271	2.007000	26279.624824
1	1629297900	2.005257	2.005659	1.999162	2.000234	113298.020353
2	1629298200	1.999123	2.003600	1.996208	2.003600	22057.611715
3	1629298500	2.004200	2.008847	2.000917	2.007030	24403.557306
4	1629298800	2.006816	2.007668	1.997762	1.997762	16562.000594
...	...	...	...	...	...	...
495	1629446100	2.448100	2.461233	2.448100	2.461233	65851.917804
496	1629446400	2.461233	2.480127	2.460000	2.469940	83735.784505
497	1629446700	2.477057	2.477057	2.459232	2.461429	23471.814504
498	1629447000	2.459592	2.461811	2.445629	2.450804	164648.046759
499	1629447300	2.448715	2.448715	2.448715	2.448715	250.000000

500 rows × 6 columns

(Stephan Avenwedde, [CC BY-SA 4.0](#))

Now calculate the EMA-10 and EMA-21 values. Luckily, pandas offers you the [ewm](#) function, which does exactly what is needed. The EMA values are added as separate columns to the DataFrame:

```
In [14]: ohlcv['ema-10'] = ohlcv['close'].ewm(span = 10, adjust=False).mean()
ohlcv['ema-21'] = ohlcv['close'].ewm(span = 21, adjust=False).mean()
```

```
In [15]: ohlcv
```

```
Out[15]:
```

	close_time	open	high	low	close	volume	ema-10	ema-21
0	1629297600	2.005271	2.007492	2.005271	2.007000	26279.624824	2.007000	2.007000
1	1629297900	2.005257	2.005659	1.999162	2.000234	113298.020353	2.005770	2.006385
2	1629298200	1.999123	2.003600	1.996208	2.003600	22057.611715	2.005375	2.006132
3	1629298500	2.004200	2.008847	2.000917	2.007030	24403.557306	2.005676	2.006213
4	1629298800	2.006816	2.007668	1.997762	1.997762	16562.000594	2.004237	2.005445
...	...	...	...	...	...	...	...	...
495	1629446100	2.448100	2.461233	2.448100	2.461233	65851.917804	2.473462	2.485005
496	1629446400	2.461233	2.480127	2.460000	2.469940	83735.784505	2.472822	2.483636
497	1629446700	2.477057	2.477057	2.459232	2.461429	23471.814504	2.470750	2.481617
498	1629447000	2.459592	2.461811	2.445629	2.450804	164648.046759	2.467124	2.478816
499	1629447300	2.448715	2.448715	2.448715	2.448715	250.000000	2.463777	2.476079

500 rows × 8 columns

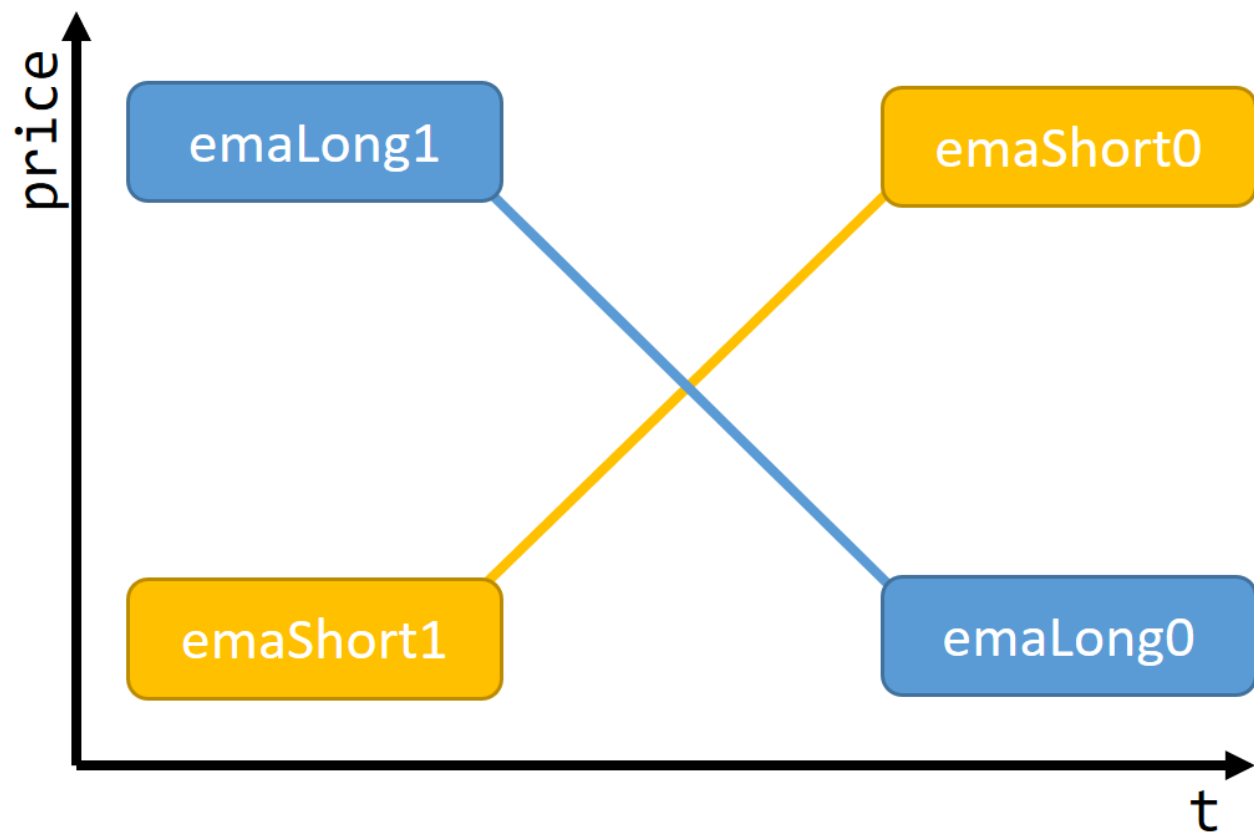
(Stephan Avenwedde, [CC BY-SA 4.0](#))

To determine if a buy or sell condition is met, you have to consider these four variables:

- **emaLong0**: Current long-term (*ema-21*) EMA value
- **emaLong1**: Last long-term (*ema-21*) EMA value (the value before emaLong0)
- **emaShort0**: Current short-term (*ema-10*) EMA value
- **emaShort1**: Last short-term (*ema-10*) EMA value (the value before emaShort0)

When the following situation comes into effect, a buy condition is met:



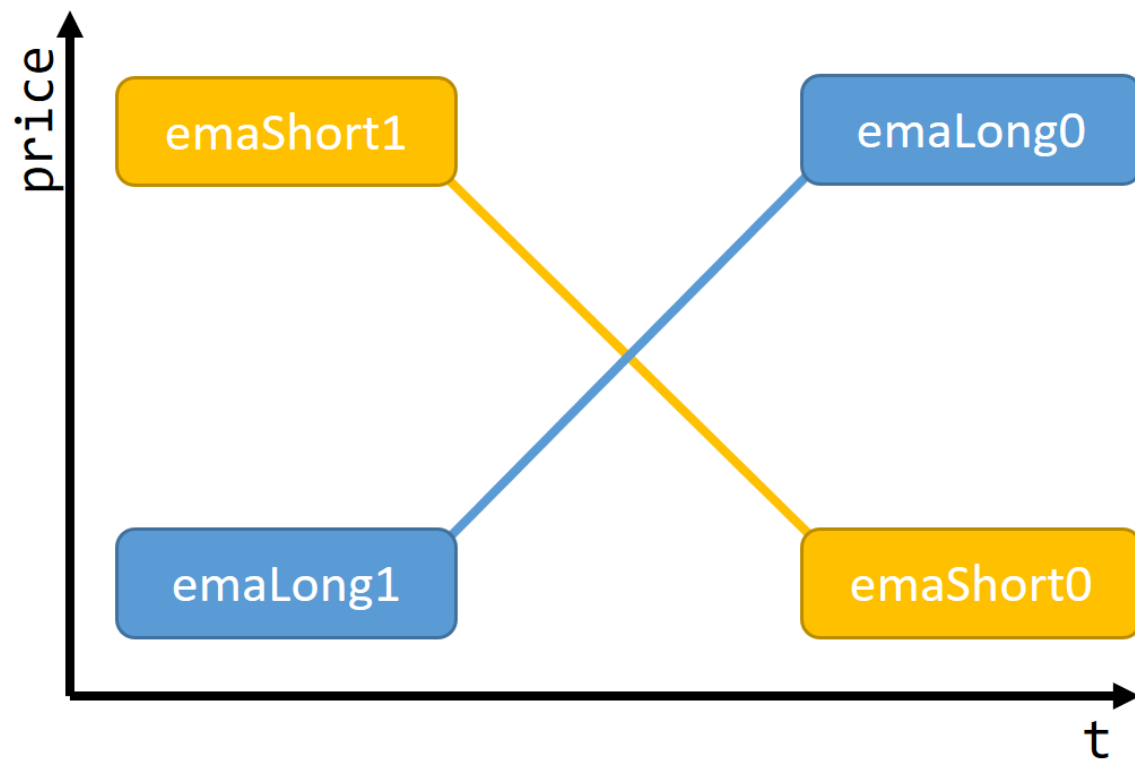


(Stephan Avenwedde, [CC BY-SA 4.0](#))

In Python code:

```
emaLong1 > emaShort1 and emaShort0 > emaLong0
```

A sell condition is met in the following situation:



(Stephan Avenwedde, [CC BY-SA 4.0](#))

In Python code:

```
emaShort1 > emaLong1 and emaLong0 > emaShort0
```

To test the DataFrame and evaluate the possible profit you could make, you could either iterate over each row and test for these conditions or, with a smarter approach, filter the dataset to only the relevant rows with built-in methods from Pandas.

Under the hood, Pandas uses [NumPy](#), which is the method of choice for fast and efficient data operation on arrays. This is, of course, convenient because the later use is to take place on a Raspberry Pi with an ARM CPU.

For the sake of clarity, the DataFrame from the example (`ADAUSD_5m.df`) with only 20 entries is used in the following examples. The following code appends a column of boolean values dependent on the condition `emaShort0 > emaLong0`:

```
In [4]: ohlcv['condition'] = ohlcv['ema-10'] > ohlcv['ema-21']
```

```
In [5]: ohlcv
```

```
Out[5]:
```

	close_time	open	high	low	close	volume	ema-10	ema-21	condition
0	1629454500	2.517279	2.526499	2.514292	2.515642	169426.266647	2.515642	2.515642	False
1	1629454800	2.517187	2.519053	2.508900	2.510031	72646.826814	2.514622	2.515132	False
2	1629455100	2.510031	2.521691	2.510031	2.520449	41457.942743	2.515681	2.515615	True
3	1629455400	2.521740	2.527249	2.519636	2.527249	34858.226532	2.517785	2.516673	True
4	1629455700	2.527249	2.534001	2.525474	2.534001	109677.296322	2.520733	2.518248	True
5	1629456000	2.534000	2.534000	2.524481	2.527657	66041.636103	2.521992	2.519104	True
6	1629456300	2.525728	2.532218	2.516475	2.520001	403971.852237	2.521630	2.519185	True
7	1629456600	2.520000	2.520870	2.510854	2.510854	17445.812775	2.519671	2.518428	True
8	1629456900	2.510854	2.510854	2.503580	2.506518	77564.463993	2.517279	2.517345	False
9	1629457200	2.506379	2.506610	2.499253	2.499576	327143.785239	2.514060	2.515730	False
10	1629457500	2.499682	2.499999	2.483484	2.489732	317949.676231	2.509637	2.513366	False
11	1629457800	2.492695	2.497164	2.484651	2.487836	61952.422092	2.505673	2.511045	False
12	1629458100	2.487380	2.499117	2.484001	2.497441	115488.500881	2.504177	2.509809	False
13	1629458400	2.497993	2.508550	2.496428	2.507713	138839.108170	2.504820	2.509618	False
14	1629458700	2.507714	2.511704	2.503317	2.510754	368660.847001	2.505899	2.509721	False
15	1629459000	2.510276	2.510784	2.492389	2.493269	153163.877641	2.503602	2.508226	False
16	1629459300	2.495581	2.501744	2.494471	2.497903	41725.153682	2.502566	2.507287	False
17	1629459600	2.497706	2.502328	2.492438	2.492438	41735.517146	2.500725	2.505937	False
18	1629459900	2.492360	2.492360	2.483197	2.483616	87220.914936	2.497614	2.503908	False
19	1629460200	2.486802	2.486802	2.486802	2.486802	50.000000	2.495648	2.502353	False

(Stephan Avenwedde, [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/))

The place of interest is when a *False* switches to *True* (buy) or when *True* switches to *False*. To filter them apply a [diff](#) operation to the *condition* column. The `diff` operation calculates the difference between the current and the previous line. In terms of boolean values, it results in:

- *False diff False* = *False*
- *False diff True* = *True*
- *True diff True* = *False*
- *True diff False* = *True*

With the following code, you apply the `diff` operation as a filter to the *condition* column without modifying it:

```
In [6]: ohlcv[(ohlcv['condition']).diff().fillna(False)]
```

```
Out[6]:
```

	close_time	open	high	low	close	volume	ema-10	ema-21	condition
2	1629455100	2.510031	2.521691	2.510031	2.520449	41457.942743	2.515681	2.515615	True
8	1629456900	2.510854	2.510854	2.503580	2.506518	77564.463993	2.517279	2.517345	False

(Stephan Avenwedde, [CC BY-SA 4.0](#))

As a result, you get the desired data: The first row (index 2) signalizes a buy condition and the second row (index 8) signalizes a sell condition. As you now have an efficient way of extracting relevant data, you can calculate possible profit.

To do so, you have to iterate through the rows and calculate the possible profit based on simulated trades. The variable `bBought` saves the state if you already bought, and `buyPrice` stores the price you bought between the iterations. You also skip the first sell indicator as it doesn't make sense to sell before you've even bought.

```
profit = 0.0
buyPrice = 0.0

bBought = False
for index, row, in trades.iterrows():
    # skip first sell-indicator
    if not row['condition'] and not bBought:
        continue
    # buy-indication
    if row['condition'] and not bBought:
        bBought = True
        buyPrice = row['close']
    # sell-indication
    if not row['condition'] and bBought:
        bBought = False
        sellPrice = row['close']

orderProfit = (sellPrice * 100) / buyPrice - 100
profit += orderProfit
```

Your one-trade mini dataset would provide you the following profit:

```
In [21]: trades = ohlcv[(ohlcv['condition']).diff().fillna(False)]
```

```
In [38]: profit = 0.0
buyPrice = 0.0
bBought = False

for index, row, in trades.iterrows():

    # skip first sell-indicator
    if not row['condition'] and not bBought:
        continue

    # buy-indication
    if row['condition'] and not bBought:
        bBought = True
        buyPrice = row['close']

    # sell-indication
    if not row['condition'] and bBought:
        bBought = False
        sellPrice = row['close']

    orderProfit = (sellPrice * 100) / buyPrice - 100

    profit += orderProfit

profit
```

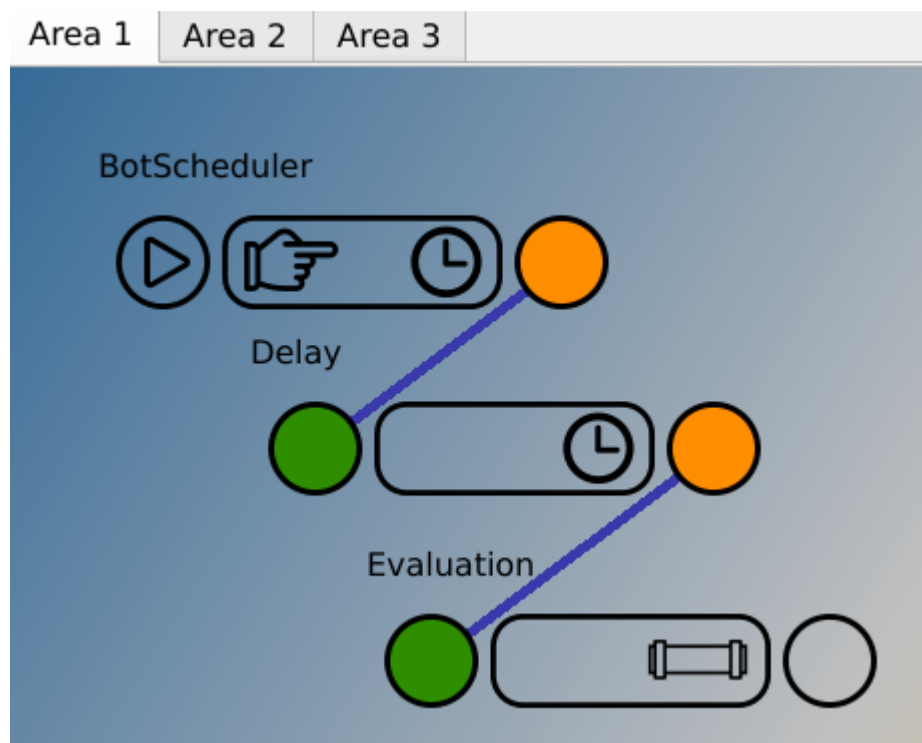
```
Out[38]: -0.5527189798325765
```

(Stephan Avenwedde, [CC BY-SA 4.0](#))

**Note:** As you can see, the strategy would have given a terrible result as you would have bought at \$2.5204 and sold at \$2.5065, causing a loss of 0.55% (order fees not included). However, this is a real-world scenario: One strategy does not work for each scenario. It is on you to find the most promising parameters (for example, using OHLCV on an hourly basis would make more sense in general).

## Implementation

You can find the implementation of the decision on the **Area 1** tab.



(Stephan Avenwedde, [CC BY-SA 4.0](#))

It implements the following functionality:

- **BotScheduler:** Same as the AcquisitionScheduler: Trigger subsequent elements every five minutes
- **Delay:** Delay the execution for 30 seconds to make sure that the latest OHLCV data was written to file
- **Evaluation:** Make the trading decision based on the EMA crossover strategy

You now know how the decision makings work, so you can take a look at the actual implementation. Open the file `generic_pipe_29dfc189.py`. It corresponds to the **Evaluation** element on the screen:

```
@dataclass
class OrderRecord:
    orderType:      bool # True = Buy, False = Sell
    price:          float # close price
    profit:         float # profit in percent

    profitCumulative: float # cumulative profit in percent
class OrderType(Enum):
    Buy = True
    Sell = False
class Element(Function):
```

```

def __init__(self, id, config, inputData, return_queue, cmd_queue):
    super().__init__(id, config, inputData, return_queue, cmd_queue)
    def execute(self):
        ### Load data ###
        file_path = Path.home() / 'Pythonic' / 'executables' / 'ADAUSD_5m.df'
        # only the last 21 columns are considered
        self.ohlc = pd.read_pickle(file_path)[-21:]
        self.bBought = False
        self.lastPrice = 0.0
        self.profit = 0.0
        self.profitCumulative = 0.0
        self.price = self.ohlc['close'].iloc[-1]
        # switches for simulation
        self.bForceBuy = False
        self.bForceSell = False
        # load trade history from file
        self.trackRecord = ListPersist('track_record')
        try:
            lastOrder = self.trackRecord[-1]
            self.bBought = lastOrder.orderType
            self.lastPrice = lastOrder.price
            self.profitCumulative = lastOrder.profitCumulative
        except IndexError:
            pass
        ### Calculate indicators ###
        self.ohlc['ema-10'] = self.ohlc['close'].ewm(span = 10, adjust=False).mean()
        self.ohlc['ema-21'] = self.ohlc['close'].ewm(span = 21, adjust=False).mean()
        self.ohlc['condition'] = self.ohlc['ema-10'] > self.ohlc['ema-21']
        ### Check for Buy- / Sell-condition ###
        tradeCondition = self.ohlc['condition'].iloc[-1] !=
        self.ohlc['condition'].iloc[-2]
        if tradeCondition or self.bForceBuy or self.bForceSell:
            orderType = self.ohlc['condition'].iloc[-1] # True = BUY, False = SELL
            if orderType and not self.bBought or self.bForceBuy: # place a buy order
                msg = 'Placing a Buy-order'
                newOrder = self.createOrder(True)
            elif not orderType and self.bBought or self.bForceSell: # place a sell order
                msg = 'Placing a Sell-order'
                sellPrice = self.price
                buyPrice = self.lastPrice
                self.profit = (sellPrice * 100) / buyPrice - 100
                self.profitCumulative += self.profit

            newOrder = self.createOrder(False)
        else: # Something went wrong
            msg = 'Warning: Condition for {}-order met but bBought is
            {}'.format(OrderType(orderType).name, self.bBought)
            newOrder = None

```

```

recordDone = Record(newOrder, msg)
self.return_queue.put(recordDone)
def createOrder(self, orderType: bool) -> OrderRecord:
newOrder = OrderRecord(
orderType=orderType,
price=self.price,
profit=self.profit,
profitCumulative=self.profitCumulative
)
self.trackRecord.append(newOrder)
return newOrder

```

As the general process is not that complicated, I want to highlight some of the peculiarities:

### Input data

The trading bot only processes the last 21 elements as this is the range you consider when calculating the exponential moving average:

```
self.ohlcv = pd.read_pickle(file_path)[-21:]
```

### Track record

The type *ListPersist* is an extended Python list object that writes itself to the file system when modified (when elements get added or removed). It creates the file `track_record.obj` under `~/Pythonic/executables/` once you run it the first time.

```
self.trackRecord = ListPersist('track_record')
```

Maintaining a track record helps to keep the state of recent bot activity.

### Plausibility

The algorithm outputs an object of the type *OrderRecord* in case conditions for a trade are met. It also keeps track of the overall situation: For example, if a buy signal was received, but `bBought` indicates that you already bought before, something must've gone wrong:

```

else: # Something went wrong

msg = 'Warning: Condition for {}-order met but bBought is
{}'.format(OrderType(orderType).name, self.bBought)
newOrder = None

```

In this scenario, *None* is returned with a corresponding log message.

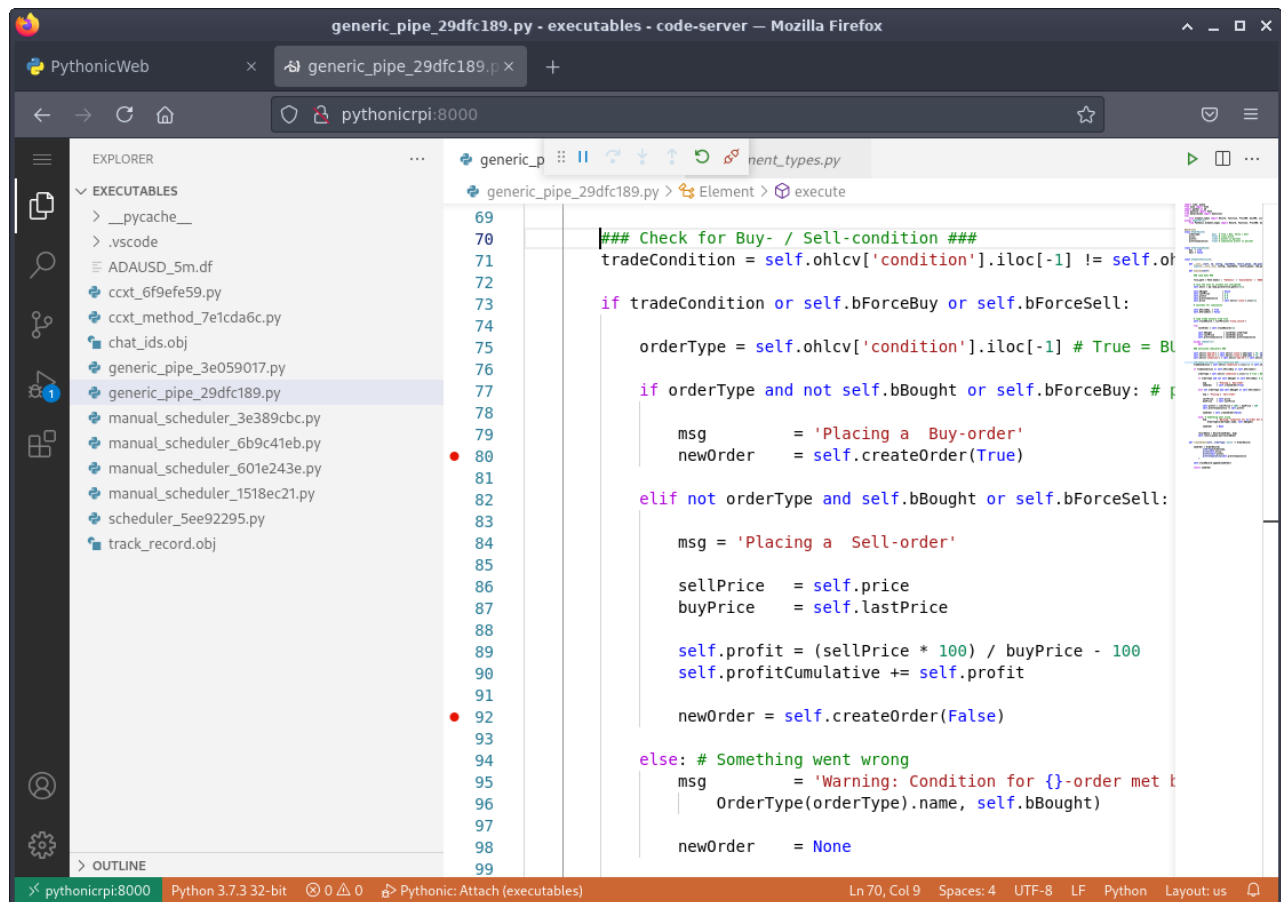


## Simulation

The Evaluation element (`generic_pipe_29dfc189.py`) contains these switches which enable you to force the execution of a buy or sell order:

```
self.bForceBuy = False
self.bForceSell = False
```

Open the code server IDE (<http://PythonicRPI:8000/>), load `generic_pipe_29dfc189.py` and set one of the switches to `True`. Attach with the debugger and add a breakpoint where the execution path enters the *inner if* conditions.

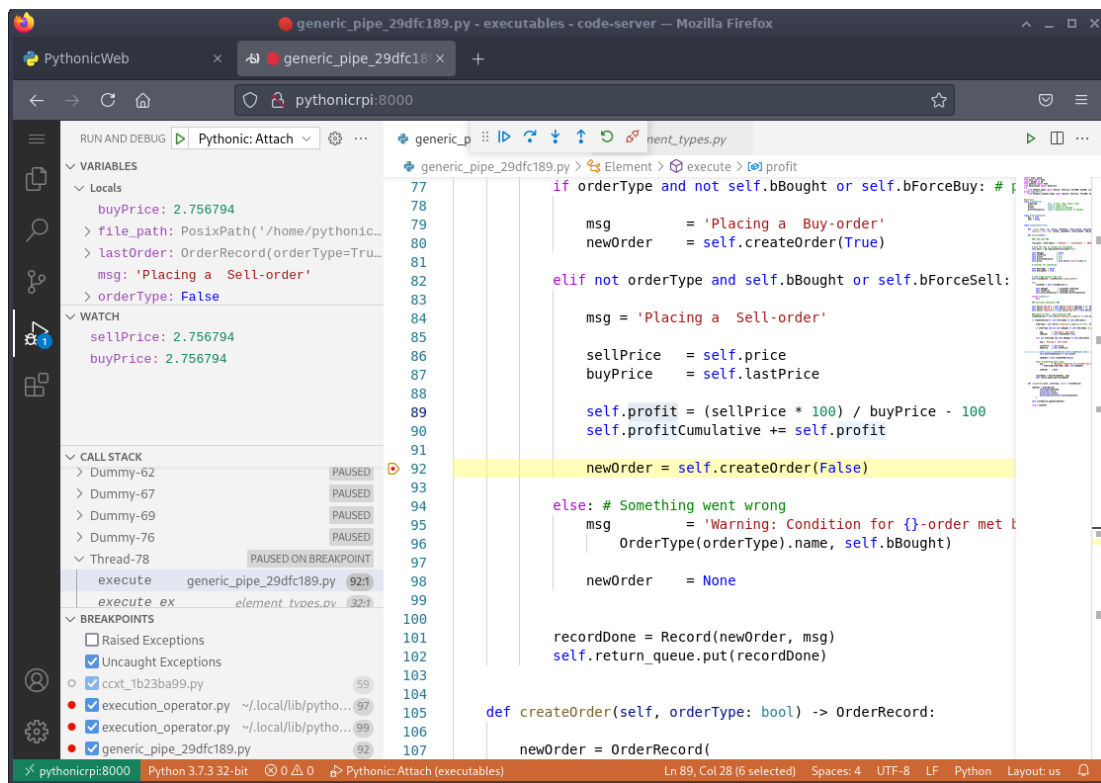


(Stephan Avenwedde, [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/))

Now open the programming GUI, add a **ManualScheduler** element (configured to *single fire*) and connect it directly to the **Evaluation** element to trigger it manually:

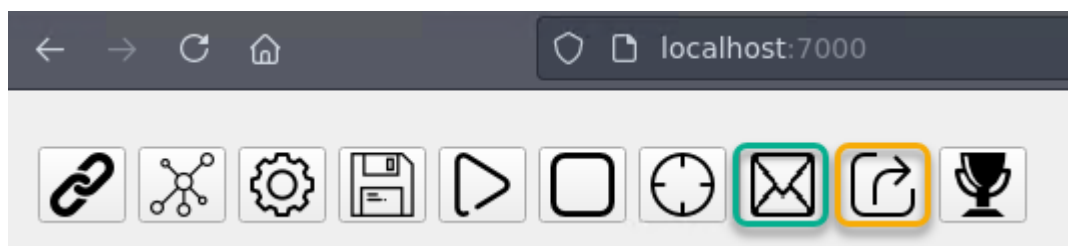
(Stephan Avenwedde, [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/))

Click the play button. The **Evaluation** element is triggered directly, and the debugger stops at the previously set breakpoint. You are now able to add, remove, or modify orders from the track record manually to simulate certain scenarios:



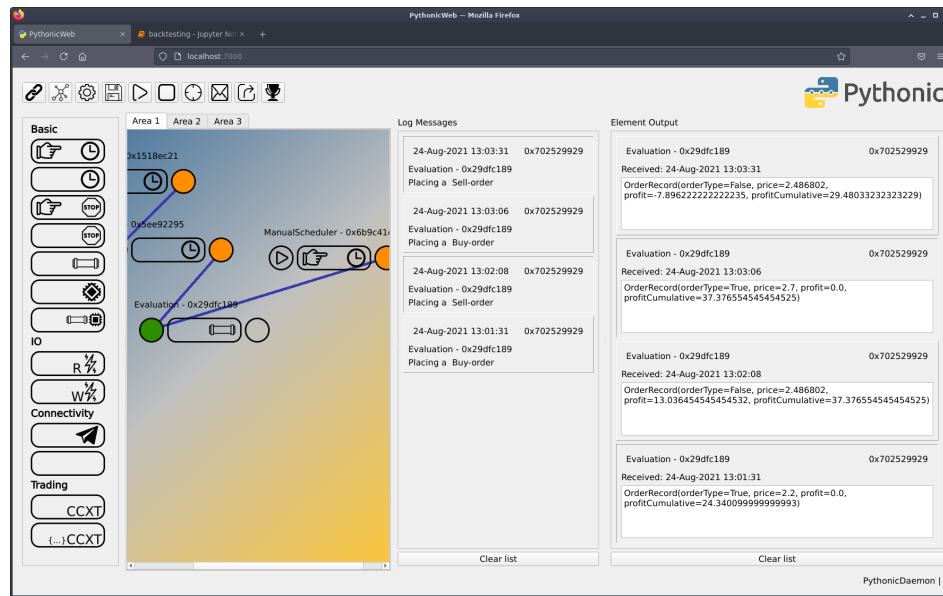
(Stephan Avenwedde, [CC BY-SA 4.0](#))

Open the log message window (green outlined button) and the output data window (orange outlined button):



(Stephan Avenwedde, [CC BY-SA 4.0](#))

You see the log messages and output of the **Evaluation** element, and the behavior of the decision-making algorithm based on your input:



(Stephan Avenwedde, [CC BY-SA 4.0](#))

## Summary

The example stops here. The final implementation could notify the user about a trade indication, place an order on an exchange, or query the account balance in advance. At this point, you should feel that everything connects and be able to proceed on your own.

Using Pythonic as a base for your trading bot is a good choice because it runs on a Raspberry Pi, is entirely accessible by a web browser, and already has logging features. It is even possible to stop on a breakpoint without disturbing the execution of other tasks using Pythonic's multiprocessing capabilities.



Stephan is a technology enthusiast who appreciates open source for the deep insight of how things work.

Stephan works as a full time support engineer in the mostly proprietary area of industrial automation software. If possible, he works on his Python-based open source projects, writing articles, or driving motorbike.

# Manage Linux users' home directories with systemd-homed

By David Both

The entire systemd concept and implementation have introduced many changes since it began to replace the old SystemV startup and init tools. Over time, systemd has been extended into many other segments of the Linux environment.

One relatively new service, systemd-homed, extends the reach of systemd into the management of users' home directories. The feature enforces human user access only and restricts system users in the User ID (UID) range between 0 and 999. I support the [systemd plan to take over the world](#), but I wondered if this was a bit excessive. Then I did some research.

## What is systemd-homed?

The systemd-homed service supports user account portability independent of the underlying computer system. A practical example is to carry around your home directory on a USB thumb drive and plug it into any system which would automatically recognize and mount it. According to Lennart Poettering, lead developer of systemd, access to a user's home directory should not be allowed to anyone unless the user is logged in. The systemd-homed service is designed to enhance security, especially for mobile devices such as laptops. It also seems like a tool that might be useful with [containers](#).

This objective can only be achieved if the home directory contains all user metadata. The `~/.identity` file stores user account information, which is only accessible to systemd-homed when the password is entered. This file holds all of the account metadata, including everything Linux needs to know about you, so that the home directory is portable to any Linux host that

uses `systemd-homed`. This approach prevents having an account with a stored password on every system you might need to use.

The home directory can also be encrypted using your password. Under `systemd-homed`, your home directory stores your password with all of your user metadata. Your encrypted password is not stored anywhere else thus cannot be accessed by anyone. Although the methods used to encrypt and store passwords for modern Linux systems are considered to be unbreakable, the best safeguard is to prevent them from being accessed in the first place. Assumptions about the invulnerability of their security have led many to ruin.

This service is primarily intended for use with portable devices such as laptops. Poettering states, "Homed is intended primarily for client machines, i.e., laptops and thus machines you typically ssh from a lot more than ssh to, if you follow what I mean." It is not intended for use on servers or workstations that are tethered to a single location by cables or locked into a server room.

The `systemd-homed` service is enabled by default on new installations—at least for Fedora, which is the distro that I use. This configuration is by design, and I don't expect that to change. User accounts are not affected or altered in any way on systems with existing filesystems, upgrades or reinstallations that keep the existing partitions, and logical volumes.

## Creating controlled users

Traditional tools such as `useradd` create accounts and home directories that `systemd-homed` does not manage. Therefore, if you continue to use the conventional user management tools, the home directories on your home directories are not managed by `systemd-homed`. This is also the case with the non-root user account created during a new installation.

## The `homectl` command

The `homectl` command creates user accounts that `systemd-homed` manages. Using the `homectl` command to create a new account generates the metadata needed to make the home directory portable.

The `homectl` command man page has a good explanation of the objectives and function of the `systemd-homed` service. However, reading the `homectl` man page is quite interesting, especially the Example section. Of the five examples, three show how to create user accounts

with specific limits imposed, such as a maximum number of concurrent processes or a maximum amount of disk space.

In a non-homectl setup, the `/etc/security/limits.conf` file imposes these limits. The only advantage I can see to this is that it adds a user and applies the limits with a single command. With the traditional method, the sysadmin must configure the `limits.conf` file manually.

## Limitations

The only significant limitation I am aware of is that it is not possible to access a user home directory remotely using OpenSSH. This limitation is due to the current inability of [PAM](#) to provide access to a home directory managed by `homectl`. Poettering seems doubtful that this can be overcome. This issue would prevent me from using `systemd-homed` for my home directory on my primary workstation or even my laptop. I typically log into both computers remotely several times per day using SSH, so this is a showstopper for me.

The other concern I can see is that you still need a Linux computer for use with a USB thumb drive with your home directory on it, and that computer needs to have `systemd-homed` running.

## It is optional

You don't have to use it, however. I plan to continue using the traditional tools for user management to support my workflow. The default for the few distros I have some little knowledge of, including Fedora, is for the `systemd-homed` service to be enabled and running. *You can disable and stop the `systemd-homed` service without impacting traditional user accounts.*

## Final thoughts

Sysadmins can use the `systemd-homed` service for a secure form of management of roaming users' home directories. It is useful on portable devices like laptops and can be especially useful for users who carry a thumb drive containing only their home directories to plug it into any convenient Linux computer.

The primary limitation of using `systemd-homed` is that it is impossible to log in remotely using SSH. And even though the `systemd-homed` is enabled by default, it does not affect home

directories created with the `useradd` command. I do need to point out that—like many systemd tools—systemd-homed is optional. So I just stopped and disabled the service.

If I need to take my home directory in a package smaller than my laptop, I can just use a live USB with persistent storage.



David Both is an Open Source Software and GNU/Linux advocate, trainer, writer, and speaker who lives in Raleigh North Carolina. He is a strong proponent of and evangelist for the "Linux Philosophy."

David has been in the IT industry for nearly 50 years. He has taught RHCE classes for Red Hat and has worked at MCI Worldcom, Cisco, and the State of North Carolina. He has been working with Linux and Open Source Software for over 20 years.

David prefers to purchase the components and build his own computers from scratch to ensure that each new computer meets his exacting specifications. His primary workstation is an ASUS TUF X299 motherboard and an Intel i9 CPU with 16 cores (32 CPUs) and 64GB of RAM in a ThermalTake Core X9 case.

David has written articles for magazines including, Linux Magazine, Linux Journal. His article "Complete Kickstart," co-authored with a colleague at Cisco, was ranked 9th in the Linux Magazine Top Ten Best System Administration Articles list for 2008. David currently writes prolifically for Opensource.com and Enable Sysadmin.

David currently has two books published: "The Linux Philosophy for SysAdmins." and "Using and Administering Linux: Zero to SysAdmin," a self-study training course in three volumes.

David can be reached at [LinuxGeek46@both.org](mailto:LinuxGeek46@both.org) or on Twitter [@LinuxGeek46](https://twitter.com/LinuxGeek46).

# An open source developer's guide to systems programming

By Alex Bunardzic

Programming is an activity that helps implement a model. What is a model? Typically, programmers model real-world situations, such as online shopping.

When you go shopping in the real world, you enter a store and start browsing. When you find items you'd like to purchase, you place them into the shopping cart. Once your shopping is done, you go to the checkout, the cashier tallies up all the items, and presents you with the total. You then pay and leave the store with your newly purchased items.

Thanks to the advancements in technology, you can now accomplish the same shopping activities without traveling to a physical store. You achieve that convenience by having a team of software creators model actual shopping activities and then simulate those activities using software programs.

Such programs run on information technology systems composed of networks and other computing infrastructure. The challenge is to make a reliable system in the presence of failures.

## Why failures?

The only way to offer virtual capabilities such as online shopping is to implement the model on a network (i.e., the internet). One problem with networks is that they are inherently unreliable. Whenever you plan to implement a network app, you must consider the following pervasive problems:

- The network is not reliable.
- The latency on the network is not zero.
- The bandwidth on the network is not infinite.



- The network is not secure.
- Network topology tends to change.
- The transport cost on the network is not zero.
- The network is not homogenous.
- "Works on my machine" is not a proof that the app is actually functional.

As can be seen from the above list, there are many reasons to expect failures when planning to launch an app or service.

## **What is a system?**

You depend on a system to support the app. So, what is a system?

A system is something that stands together, meaning it's a composition of programs that offer services to other programs. Such a design is loosely coupled. It is distributed and decentralized (i.e., it does not have global supervision/management).

## **What is a reliable system?**

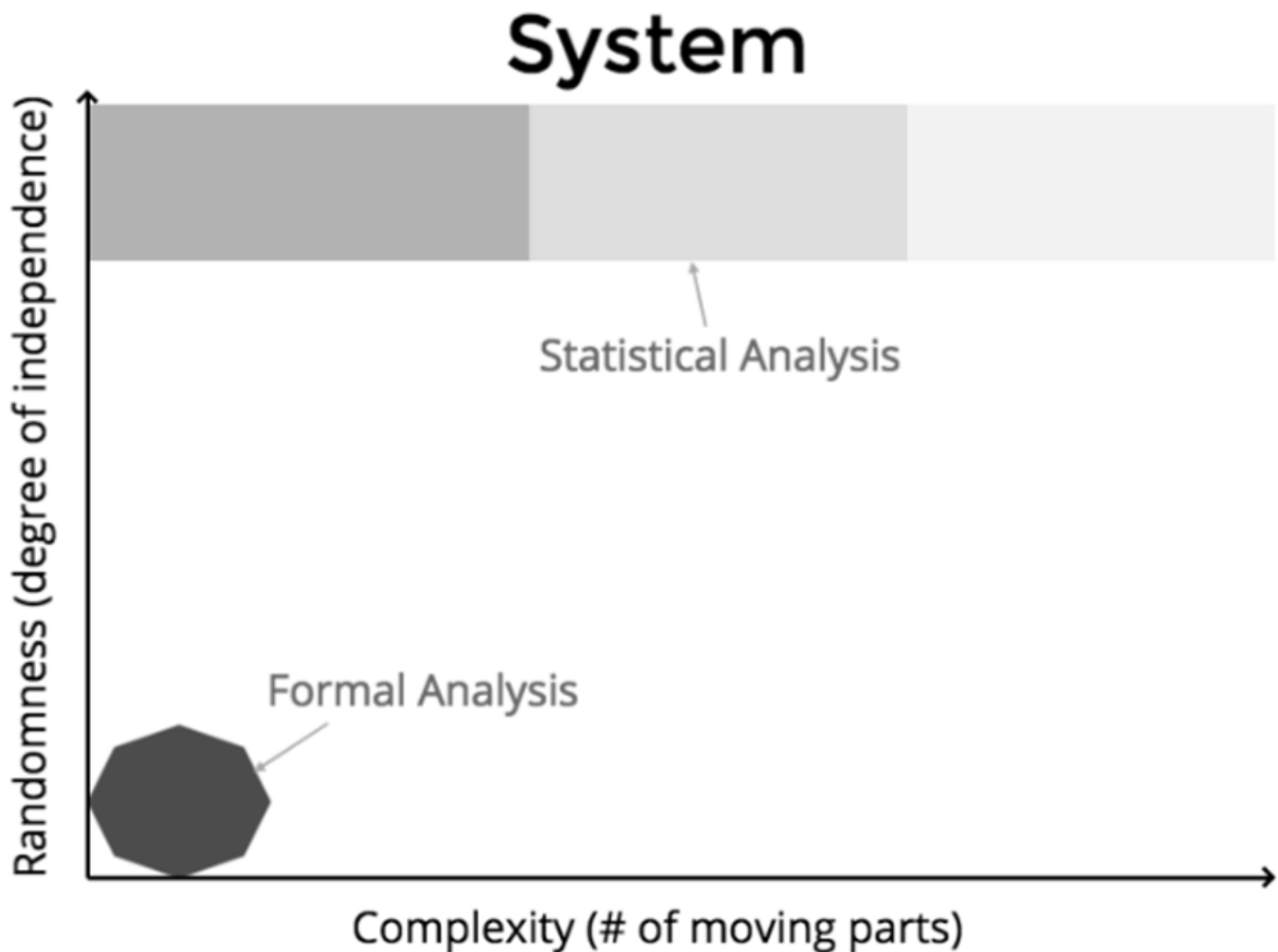
Consider the attributes that make up a reliable system:

- A reliable system is a system that is always up and running. Such a system is capable of graceful degradation, meaning that when performance starts to degrade, the system will not suddenly stop working.
- A reliable system is not only always up and running, but it is also capable of progressive enhancement. As the demand for the system's capabilities increases, a reliable system scales to meet the needs.
- A reliable system is also easily maintainable without expensive changes.
- A reliable system is low-risk. It is safe and simple to deploy changes to such a system, either by rolling back or forward.

## **Everything built eventually exceeds the ability to understand it**

Every successful system was created from a much simpler design. As systems are enhanced and embellished, they eventually reach a point where their complexity cannot be easily understood.

Consider a system that consists of many moving parts. As the number of moving parts in the system increases, the degree of interdependence between those moving parts also increases (Figure 1).



(Alex Bunardzic, CC BY-SA 4.0)

It is only during the early stages of the growth of that system that people can perform a formal analysis of the system. After a certain point of system complexity, humans can only reason about the system by applying statistical analysis.

There is a gap between formal analysis and statistical analysis (Figure 2).

(Alex Bunardzic, CC BY-SA 4.0)

## How to program a system?

Developers know how to write useful apps, but they must also know how to program a system that enables the app to function on the network.

It turns out that there doesn't seem to be a system programming language available. While developers may know many programming languages (e.g., C, Java, C++, C#, Python, Ruby, JavaScript, etc.), all those languages specialize in modeling and emulating the functioning of an app. But how does one model system functionality?

Look at how the system is assembled. Basically, in a system, programs talk to each other. How do they do that?

They communicate over a network. Since there cannot be a system without two or more programs talking to each other, it is clear that the only way to program a system is to program a network.

Before looking more closely at how to program a network, I will examine the main problem with networks—failure.

## Failures are at the system level

How do failures occur in a system? One way is when one or more programs suddenly becomes unavailable.

That failure has nothing to do with programming errors. Actually, programming errors are not really errors—they are bugs to be squashed!

A network is basically a chain, and as everyone knows, a chain is only as strong as its weakest link.

(Alex Bunardzic, CC BY-SA 4.0)

When a link breaks (i.e., when one of the programs becomes unavailable), it is critical to prevent that outage from bringing the entire system down.

How do administrators do that? They provide an abstraction boundary that stops the propagation of errors. I will now examine ways to provide such an abstraction boundary inside the system. Doing that amounts to programming a system.

## Best practices in system programming

It is very important to design programs and services to meet the needs of the machines. It is a common mistake to create programs and services to serve human needs. But when doing systems programming, such an approach is incorrect.

There is a fundamental difference between designing services for machines versus humans. Machines do not need operational interfaces. However, humans cannot consume services without a functional interface.

What machines need is programming interfaces. Therefore, when doing systems programming, focus entirely on the application programming interfaces (APIs). It will be easy to bolt operational interfaces on top of the already implemented programming interfaces, so do not rush into creating operational interfaces first.

It is also important to build only simple services. This may seem unreasonable at first, but once you understand that simple services are easily composable into more complex ones, it makes more sense.

Why are simple services so essential when doing systems programming? By focusing on simple services, developers minimize the risk of slipping into premature abstraction. It becomes impossible to over-abstract such simple services. The result is a system component that is easy to make, reason about, deploy, fix, and replace.

Developers must avoid the temptation to turn the service into a monolith. Abstain from doing that by refusing to add functionalities and features. Furthermore, resist turning service into a stack. When other users (programs) decide to use the services the component offers, they should be free to choose commodities suitable for the consumption of the services.

Let the service users decide which datastore to use, which queue, etc. Programmers must never dictate a custom stack to clients.

Services must be fully isolated and independent. In other words, services must remain autonomous.

### Value of values

What is a value in the context of programming? The following attributes characterize a value:

- No identity
- Ephemeral

- Nameless
- On the wire

Consider an example value of a service that returns the total monthly service charge. Suppose a customer receives \$425.00 as a monthly service charge. What are the characteristics of the value \$425.00?

- It has no identity.
- No name – it is just four hundred twenty-five dollars—no need for a separate name.
- It is ephemeral – as the time progresses, the monthly charge keeps changing.
- It is always sent on a wire and received by the client.

The ephemeral nature of values implies flow.

## **Systems are not place-oriented**

A place-oriented product could be depicted as a ship being built in a shipyard.



(Alex Bunardzic, CC BY-SA 4.0)

## Systems are flow-oriented



(Alex Bunardzic, CC BY-SA 4.0)

For example, cars are built on a moving assembly line.

### How do values flow in the system?

Values undergo transformations and are moved, routed, and recorded.

- Transform
- Move
- Route
- Record
- Keep the above activities segregated

How do values move in the system?

- Source => destination
- Mover (producer) depends on identity/availability
- Must decouple producers from consumers
- Must remove dependency on identity
- Must remove dependency on availability



- Use queues Pub/sub

It is essential to avoid dependencies for values to flow effectively through the system. Brittle designs include processes that count on a certain service being found by its identity or requiring a certain service to be available. The only robust design that allows values to flow through the system is using queues to decouple dependencies. It is recommended to use the publish/subscribe queuing model.

## **Design services primarily for machines**

Avoid designing services to be consumed by humans. Machines should never be expected to access services via operational interfaces. Build human operational interfaces only after you've built a machine-centric service.

Strive to build only simple services. Simple services are easily composable. When designing simple services, there is no danger of premature abstraction.

It is not possible to over-abstract a simple service.

## **Avoid turning a service into a monolith**

Abstain from adding functionality and features (keep it super simple). Avoid at all costs turning a service into a stack. Allow service users to choose which commodities to use when consuming them. Let them decide which datastore to use, which queue, etc. Don't dictate your custom stack to clients.

## **System failure model is the only failure model**

Next, acknowledge that system failures are guaranteed to happen! It is not the question of if but when and how often.

When do exceptions occur? Any time a runtime system doesn't know what to do, the result is an exception and a system failure.

Those failures are different from programming errors. The errors occur when a team makes mistakes while implementing the processing logic (developers call those errors "bugs").

Whenever a system fails, notice that it is partial and uncoordinated. It is improbable that the entire system would fail at once (almost impossible for such an event to happen).

# Minimum requirements for reliable systems

At a minimum, a reliable system must possess the following capabilities:

- Concurrency
- Fault encapsulation
- Fault detection
- Fault identification
- Hot code upgrade
- Stable storage
- Asynchronous message passing

I'll examine those attributes one by one.

## Concurrency

For the system to be capable of handling two or more processes concurrently, it must be non-imperative. The system must never block the processing or apply the "pause" button on the process. Furthermore, the system must never depend on a shared mutable state.

In a concurrent system, everything is a process. Therefore, it is paramount that a reliable system must have a lightweight mechanism for creating parallel processes. It also must be capable of efficient context switching between processes and message passing.

Any process in a concurrent system must rely on fault detection primitives to be able to observe another process.

## Fault encapsulation

Faults that occur in one process must not be able to damage/impair other processes in the system.

"The process achieves fault containment by sharing no state with other processes; its only contact with other processes is via messages carried by a kernel message system." - Jim Gray

Here is another useful quote from Jim Gray:

"As with hardware, the key to software fault-tolerance is to hierarchically decompose large systems into modules, each module being a unit of service and a unit of failure. A failure of a module does not propagate beyond the module."

To achieve fault tolerance, it is necessary to only write code that handles the normal case.



In case of a failure, the only recommended course of action is to let it crash! It is not a good practice to fix the failure and continue. A different process should handle any error (the escalation error handling model).

It is crucial to constantly ensure clean separation between error recovery code and normal case code. Doing so greatly simplifies the overall system design and system architecture.

## **Fault detection**

A programming language must be able to detect exceptions both locally (in the process where the exception occurred) and remotely (seeing that an exception occurred in a non-local process).

A component is considered faulty once its behavior is no longer consistent with its specification. Error detection is an essential component of fault tolerance.

Try to keep tasks simple to increase the likelihood of success.

In the face of failure, administrators become more interested in protecting the system against damage than offering full service. The goal is to provide an acceptable level of service and become less ambitious when things start to fail.

Try to perform a task. If you cannot perform a task, try to perform a simpler task.

## **Fault identification**

You should be able to identify why an exception occurred.

## **Hot code upgrade**

The ability to change code as it is executing and without stopping the system.

## **Stable storage**

Developers need a stable error log that will survive a crash. Store data in a manner that survives a system crash.

## **Asynchronous message passing**

Asynchronous message passing should be the default choice for inter-service communication.

## Well-behaved programs

A system should be composed of well-behaved programs. Such programs should be isomorphic to the specification. If the specification says something silly, then the program must faithfully reproduce any errors in the specification. If the specification doesn't say what to do, raise an exception!

Avoid guesswork—this is not the time to be creative.

"It is essential for security to be able to isolate mistrusting programs from one another, and to protect the host platform from such programs. Isolation is difficult in object-oriented systems because objects can easily become aliased (i.e., at least two or more objects hold a reference to an object)" -Ciaran Bryce

Tasks cannot directly share objects. The only clean way for tasks to communicate is to use a standard copying communication mechanism.

## Wrap up

Applications run on systems and understanding how to properly program systems is a critical skill for developers. Systems include reliability and complexity that are best managed using a series of best practices. Some of these include:

- Processes are units of fault encapsulation.
- Strong isolation leads to autonomy.
- Processes do what they are supposed to do or fail as soon as possible (fail fast).
- Allowing components to crash and then restart leads to a simpler fault model and more reliable code. Failure, and the reason for failure, must be detectable by remote processes.
- Processes share no state, but communicate by message passing.



Alex has been doing software development since 1990. His current passion is how to bring *soft* back into *software*. Alex is a consultant at WorkSafeBC, an organization dedicated to ethical treatment of safe work environments in support of employees and employers in the province of British Columbia.

To read more of Alex's writing on technology, visit his blog at <http://digitalexprt.com/blog.html>

# What I miss about open source conferences

By Mike Bursell

A typical work year would involve my attending maybe six to eight conferences in person and speaking at quite a few of them. A few years ago, I stopped raiding random booths at the exhibitions usually associated with these for t-shirts for the simple reason that I had too many of them. That's not to say that I wouldn't accept one here or there if it was particularly nice, or an open source project which I esteemed particularly, for instance. Or ones which I thought my kids would like—they're not "cool" but are at least useful for sleepwear, apparently. I also picked up a lot of pens and enough notebooks to keep me going for a while.

And then, at the beginning of 2020, the pandemic hit, I left San Francisco, where I'd been attending meetings co-located with RSA North America (my employer at the time made the somewhat prescient decision not to allow us to go to the main conference), and I've not attended any in-person conferences since.

There are some good things about this, the most obvious being less travel, though, of late, my family has been dropping an increasing number of not-so-subtle hints about how it would be good if I let them alone for a few days so they can eat food I don't like (pizza and macaroni cheese, mainly) and watch films that I don't enjoy (largely, but not exclusively, romcoms on Disney+). The downsides are manifold. Having to buy my own t-shirts and notebooks, obviously, though it turns out that I'd squirreled away enough pens for the duration. It also turned out that the move to USB-C connectors hadn't sufficiently hit the conference swag industry by the end of 2019 for me to have enough of those to keep me going, so I've had to purchase some of those. That's the silly, minor stuff, though—what about areas where there's real impact?

Virtual conferences aren't honestly too bad, and the technology has definitely improved over the past few months. I've attended some very good sessions online (and given my share of sessions and panels, whose quality I won't presume to judge), but I've realised that I'm much more likely to attend borderline-interesting talks not on my main list of "must-sees" (some of which turn out to be very valuable) if I've actually traveled to get to a venue. The same goes for attention. I'm much less likely to be checking email, writing emails, and responding to chat

messages in an in-person conference than a virtual one. It's partly about the venue, moving between rooms, and not bothering to get my laptop out all the time—not to mention the politeness factor of giving your attention to the speaker(s) or panellists. When I'm sitting at my desk at home, none of these is relevant, and the pull of the laptop (which is open anyway to watch the session) is generally irresistible.

Two areas that have really suffered, though, are the booth experience and the "hallway track." I've had some very fruitful conversations, both from dropping by booths (sometimes mainly for a t-shirt—see above) or from staffing a booth and meeting those who visit. I've yet to attend any virtual conferences where the booth experience has worked, particularly for small projects and organisations. Online chat isn't the same, and the serendipitous aspect of wandering past a booth and seeing something you'd like to talk about is pretty much entirely missing if you have to navigate a set of webpages of menu options with actual intent.

The hallway track is meeting people outside a conference's main sessions, either people you know already or as conversations spill out of sessions you've been attending. Knots of people asking questions of presenters or panellists can reveal shared interests, opposing but thought-provoking points of view, or just similar approaches to a topic which can lead to valuable professional relationships and even long-term friendships. I'm not a particularly gregarious person—particularly if I'm tired and jetlagged—but I really enjoy catching up with colleagues and friends over a drink or a meal from time to time. While that's often difficult given the distributed nature of the companies and industries I've been involved with, conferences have presented great opportunities to meet up, have a chinwag and discuss the latest tech trends, mergers and acquisitions, and fashion failures of our fellow attendees. This is what I miss most: I can buy my own t-shirts, but friendships need nurturing. And I hope that we can safely start attending conferences again so that I can meet up with friends and share a drink. I just hope I'm not the one making the fashion mistakes (this time).



*I've been in and around Open Source since around 1997, and have been running (GNU) Linux as my main desktop at home and work since then. I'm a security bod and architect, co-founder of the [Enarx](#) project, and am currently CEO of a start-up in the Confidential Computing space. I have a blog - "[Alice, Eve & Bob](#)" - where I write (sometimes rather parenthetically) about security. I live in the UK and like single malts.*

# What you need to know about cluster logging in Kubernetes

By Mike Calizo

Server and application logging is an important facility for developers, operators, and security teams to understand an application's state running in their production environment.

Logging allows operators to determine if the applications and the required components are running smoothly and detect if something unusual is happening so they can react to the situation.

For developers, logging gives visibility to troubleshoot the code during and after development. In a production setting, the developer usually relies on a logging facility without debugging tools. Coupled with logging from the systems, developers can work hand in hand with operators to effectively troubleshoot issues.

The most important beneficiary of logging facilities is the security team, especially in a cloud-native environment. Having the ability to collect information from applications and system logs enables the security team to analyze the data from authentication, application access to malware activities where they can respond to them if needed.

Kubernetes is the leading container platform where more and more applications get deployed in production. I believe that understanding the logging architecture of Kubernetes is a very important endeavor that every Dev, Ops, and Security team needs to take seriously.

In this article, I discuss how different container logging patterns in Kubernetes work.

## System logging and application logging

Before I dig deeper into the Kubernetes logging architecture, I'd like to explore the different logging approaches and how both functionalities are critical features of Kubernetes logging.

There are two types of system components: Those that run in a container and those that do not. For example:

- The Kubernetes scheduler and `kube-proxy` run in a container.
- The `kubelet` and container runtime do not run in containers.

Similar to container logs, system container logs get stored in the `/var/log` directory, and you should rotate them regularly.

Here I consider container logging. First, I look at cluster-level logging and why it is important for cluster operators. Cluster logs provide information about how the cluster performs. Information like why pods got evicted or the node dies. Cluster logging can also capture information like cluster and application access and how the application utilizes compute resources. Overall, a cluster logging facility provides the cluster operators information that is useful for cluster operation and security.

The other way to capture container logs is through the application's native logging facility. Modern application design most likely has a logging mechanism that helps developers troubleshoot application performance issues through standard out (`stdout`) and error streams (`stderr`).

To have an effective logging facility, Kubernetes implementation requires both app and system logging components.

## 3 types of Kubernetes container logging

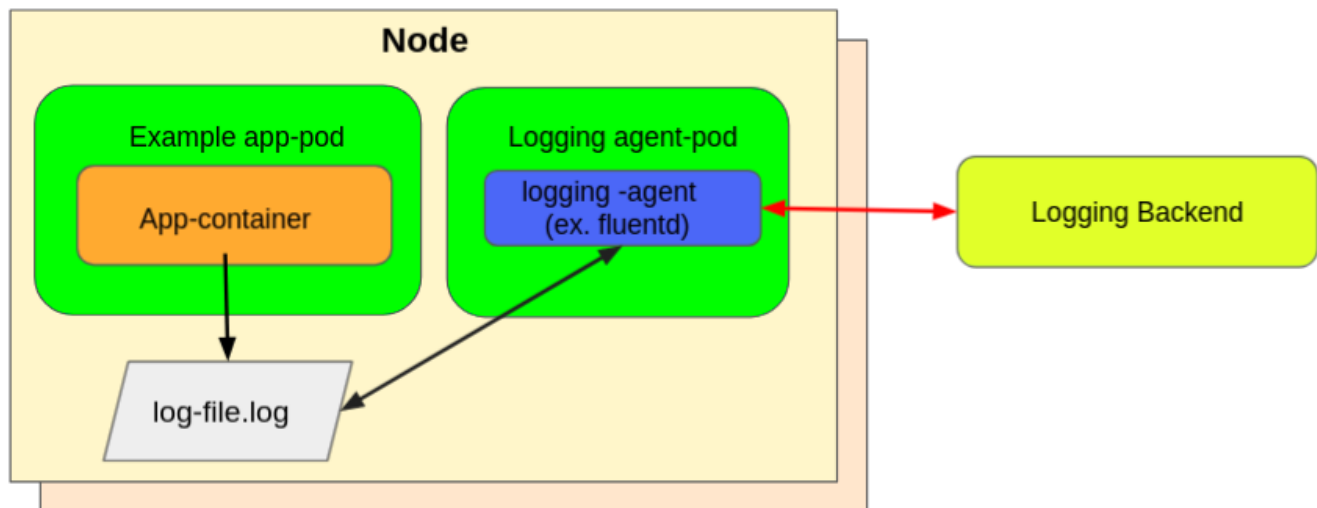
There are three prominent methods of cluster-level logging that you see in most of the Kubernetes implementations these days.

1. Node-level logging agent
2. Sidecar container application for logging
3. Exposing application logs directly to logging backend

## Node level logging agent

I'd like to consider the node-level logging agent. You usually implement these using a DaemonSet as a deployment strategy to deploy a pod (which acts as a logging agent) in all the Kubernetes nodes. This logging agent then gets configured to read the logs from all Kubernetes nodes. You usually configure the agent to read the nodes `/var/log` directory capturing `stdout/stderr` streams and send it to the logging backend storage.

The figure below shows node-level logging running as an agent in all the nodes.



(Mike Calizo, [CC BY-SA 4.0](#))

To set up node-level logging using the `fluentd` approach as an example, you need to do the following:

First, you need to create a ServiceAccount called `fluentd`. This service account gets used by the `Fluentd` Pods to access the Kubernetes API, and you need to create them in the logging Namespace with the label `app: fluentd`.

```
#fluentd-SA.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: fluentd
  namespace: logging
  labels:
    app: fluentd
```

You can view the complete example in this [repo](#).

You then need to create a ConfigMap `fluentd-configmap`. This provides a config file to the `fluentd` daemonset with all the required properties.

```
#fluentd-daemonset.yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: logging
  labels:
    app: fluentd
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      app: fluentd
      kubernetes.io/cluster-service: "true"
  template:
    metadata:
      labels:
        app: fluentd
        kubernetes.io/cluster-service: "true"
    spec:
      serviceAccount: fluentd
      containers:
        - name: fluentd
          image: fluent/fluentd-kubernetes-daemonset:v1.7.3-debian-elasticsearch7-
1.0
          env:
            - name: FLUENT_ELASTICSEARCH_HOST
              value: "elasticsearch.logging.svc.cluster.local"
            - name: FLUENT_ELASTICSEARCH_PORT
              value: "9200"
            - name: FLUENT_ELASTICSEARCH_SCHEME
              value: "http"
            - name: FLUENT_ELASTICSEARCH_USER
              value: "elastic"
            - name: FLUENT_ELASTICSEARCH_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: efk-pw-elastic
                  key: password
            - name: FLUENT_ELASTICSEARCH_SED_DISABLE
              value: "true"
          resources:
```



```

    limits:
      memory: 512Mi
    requests:
      cpu: 100m
      memory: 200Mi
  volumeMounts:
  - name: varlog
    mountPath: /var/log
  - name: varlibdockercontainers
    mountPath: /var/lib/docker/containers
    readOnly: true
  - name: fluentconfig
    mountPath: /fluentd/etc/fluent.conf
    subPath: fluent.conf
  terminationGracePeriodSeconds: 30
  volumes:
  - name: varlog
    hostPath:
      path: /var/log
  - name: varlibdockercontainers
    hostPath:
      path: /var/lib/docker/containers
  - name: fluentconfig
    configMap:
      name: fluentdconf

```

You can view the complete example in this [repo](#). Here's the code to deploy a `fluentd` `daemonset` as the log agent:

```

#fluentd-daemonset.yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: logging
  labels:
    app: fluentd
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      app: fluentd
      kubernetes.io/cluster-service: "true"
  template:
    metadata:
      labels:
        app: fluentd

```

```

    kubernetes.io/cluster-service: "true"
spec:
  serviceAccount: fluentd
  containers:
    - name: fluentd
      image: fluent/fluentd-kubernetes-daemonset:v1.7.3-debian-elasticsearch7-
1.0
      env:
        - name: FLUENT_ELASTICSEARCH_HOST
          value: "elasticsearch.logging.svc.cluster.local"
        - name: FLUENT_ELASTICSEARCH_PORT
          value: "9200"
        - name: FLUENT_ELASTICSEARCH_SCHEME
          value: "http"
        - name: FLUENT_ELASTICSEARCH_USER
          value: "elastic"
        - name: FLUENT_ELASTICSEARCH_PASSWORD
          valueFrom:
            secretKeyRef:
              name: efk-pw-elastic
              key: password
        - name: FLUENT_ELASTICSEARCH_SED_DISABLE
          value: "true"
      resources:
        limits:
          memory: 512Mi
        requests:
          cpu: 100m
          memory: 200Mi
      volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
        - name: fluentconfig
          mountPath: /fluentd/etc/fluent.conf
          subPath: fluent.conf
      terminationGracePeriodSeconds: 30
  volumes:
    - name: varlog
      hostPath:
        path: /var/log
    - name: varlibdockercontainers
      hostPath:
        path: /var/lib/docker/containers
    - name: fluentconfig
      configMap:

```

```
name: fluentdconf
```

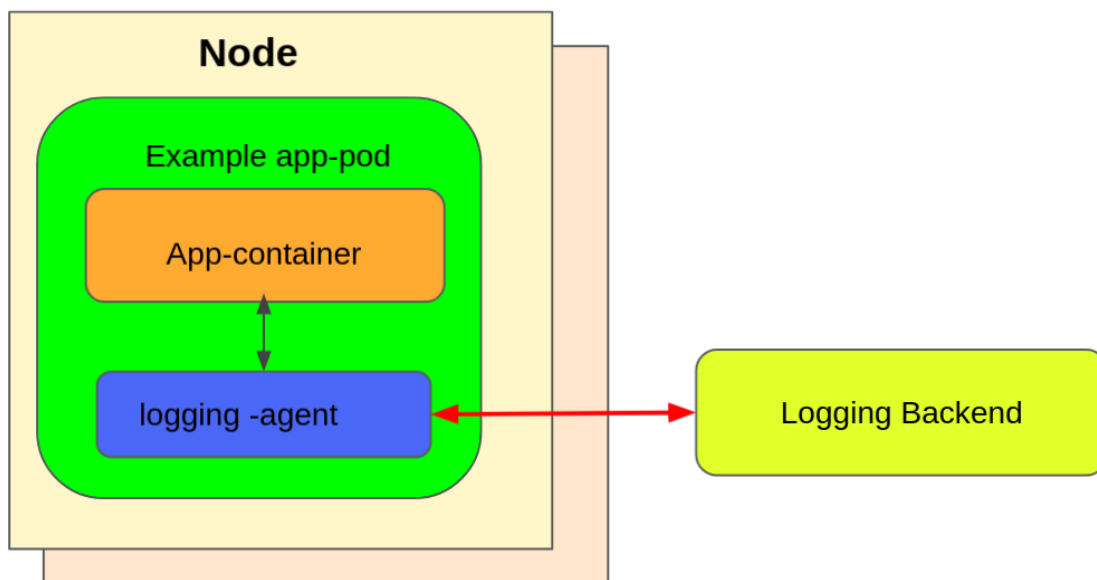
To put this together:

```
kubectl apply -f fluentd-SA.yaml \
-f fluentd-configmap.yaml \
-f fluentd-daemonset.yaml
```

## Sidecar container application for logging

The other approach is by using a dedicated sidecar container with a logging agent. The most common implementation of the sidecar container is by using [Fluentd](#) as a log collector. In the enterprise deployment (where you won't worry about a little compute resource overhead), a sidecar container using `fluentd` (or [similar](#)) implementation offers flexibility over cluster-level logging. This is because you can tune and configure the collector agent based on the type of logs, frequency, and other possible tunings you need to capture.

The figure below shows a sidecar container as a logging agent.



(Mike Calizo, [CC BY-SA 4.0](#))

For example, a pod runs a single container, and the container writes to two different log files using two different formats. Here's a configuration file for the pod:

```
#log-sidecar.yaml
apiVersion: v1
kind: Pod
```

```

metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-log
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/1.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  volumes:
  - name: varlog
    emptyDir: {}

```

To put this together, you can run this pod:

```
$ kubectl apply -f log-sidecar.yaml
```

To verify if the sidecar container works as a logging agent, you can do:

```
$ kubectl logs counter count-log
```

The expected output should look like this:

```

$ kubectl logs counter count-log-1
Thu 04 Nov 2021 09:23:21 NZDT
Thu 04 Nov 2021 09:23:22 NZDT
Thu 04 Nov 2021 09:23:23 NZDT
Thu 04 Nov 2021 09:23:24 NZDT

```

## Exposing application logs directly to logging backend

The third approach, which (in my opinion) is the most flexible logging solution for Kubernetes container and application logs, is by pushing the logs directly to the logging backend solution. Although this pattern does not rely on the native Kubernetes capability, it offers flexibility that most enterprises need like:

1. Extend a wider variety of support for network protocols and output formats.
2. Allows load balancing capability and enhances performance.
3. Configurable to accept complex logging requirements through upstream aggregation

Because this third approach relies on a non-Kubernetes feature by pushing logs directly from every application, it is outside the Kubernetes scope.

## Conclusion

The Kubernetes logging facility is a very important component for an enterprise deployment of a Kubernetes cluster. I discussed three possible patterns that are available for use. You need to find a suitable pattern for your needs.

As shown, the node-level logging using `daemonset` is the easiest deployment pattern to use, but it also has some limitations that might not fit your organization's needs. On the other hand, the sidecar pattern offers flexibility and customization that allows you to customize what type of logs to capture, giving you compute resource overhead. Finally, exposing application logs directly to the backend log facility is another enticing approach that allows further customization.

The choice is yours. You just need to find the approach that fits your organization's requirements.



Mike Calizo is the Principal Customer Success Manager of Elastic.co focused on government customers. Mike believes that "data is power" and harnessing this power can improve organizations to leverage their own insights to differentiate through innovation and drive efficiencies with cost optimization strategies. Mike is also a host of [Panzit Professional podcast](#).

# Create an app with this Arnold Schwarzenegger-themed programming language

By Jessica Cherry

Have you ever wished programming were more like an action movie? If you answered yes, then I have the language for you.

While wandering the internet to find the most obscure and fun open source languages, I came across ArnoldC. ArnoldC is an imperative programming language where the basic keywords are replaced with quotes from various Arnold Schwarzenegger movies.

For this tutorial, I'll be using a Debian-based operating system with Terminator and the Vim editor. While you follow this tutorial, I would highly recommend rewatching some older Schwarzenegger films just for fun!

## Install ArnoldC

ArnoldC is hosted in [GitHub](https://github.com). Before starting, I suggest creating a directory to hold your new project so it won't get lost. Below are my commands to get ArnoldC on your computer.

```
$ mkdir arnoldc
$ cd arnoldc/
/arnoldc$ wget http://lhartikk.github.io/ArnoldC.jar
--2022-01-16 14:11:18-- http://lhartikk.github.io/ArnoldC.jar
Resolving lhartikk.github.io (lhartikk.github.io)... \
185.199.108.153, 185.199.109.153, 185.199.110.153, ...
Connecting to lhartikk.github.io (lhartikk.github.io)\
|185.199.108.153|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 12958233 (12M) [application/java-archive]
```

Saving to: 'ArnoldC.jar'  
ArnoldC.jar

100%

## Short keyword overview

First, I'll explain some of the keywords you'll need to build an app. Keep in mind that all of these keywords need to be in all caps when writing your application.

Printing strings or variables: TALK TO THE HAND

Example: TALK TO THE HAND "hello there"

Creating a variable: GET TO THE CHOPPER

Example: GET TO THE CHOPPER var1

Setting the variable: HERE IS MY INVITATION

Example (in pattern format):

```
GET TO THE CHOPPER var1  
HERE IS MY INVITATION value1
```

Once you've finished with the assigned variable, the next line is ENOUGH TALK.

False: I LIED

True: NO PROBLEMO

Return: I'LL BE BACK

These are some of my favorite keywords from the complete list, but you can always consult the ArnoldC wiki for more.

## Hello world

I'll start with a small "hello world" app to show the ArnoldC language in use.

First, use the `echo` command to output the string "hello world" into a hello file:

```
$echo -e "IT'S SHOWTIME\nTALK TO THE HAND \"hello world\  
\"\nYOU HAVE BEEN TERMINATED" > hello.arnoldc
```

Next, use `java -jar` to create the app using ArnoldC:

```
$java -jar ArnoldC.jar hello.arnoldc
```

Then use the `java` command to run the program:

```
$ java hello  
hello world
```

If you followed these instructions, congratulations on your first under-3-minute app in a completely frivolous language.

## Let's count

In this next example, I'll have my app count to 20. The odd patterning makes this program pretty interesting.

First, create the file using Vim so you can just start writing the app: `arnoldc$ vi count.arnoldc`

Create the `begin main` with `IT'S SHOWTIME.`

Next, set up the declared variable: `HEY CHRISTMAS TREE isLessThan20`

Then, set the initial value of the variable to true, making that required: `YOU SET US UP @NO PROBLEMO`

Repeat these steps with the variable `n` and make the first set value 0:

```
HEY CHRISTMAS TREE n  
YOU SET US UP 0
```

From here, move into a while loop with the first variable: `STICK AROUND isLessThan20`

Assign the variable to look at: `GET TO THE CHOPPER n`

Then set the value to plus one:

```
HERE IS MY INVITATION n  
GET UP 1
```

Moving on to ending the assigned variable: `ENOUGH TALK`

Print the number: `TALK TO THE HAND n`



Look at the assigned variable again, then set the variable to 20:

```
GET TO THE CHOPPER isLessThan20  
HERE IS MY INVITATION 20
```

Check to see if the number is less than 20: LET OFF SOME STEAM BENNET n

Moving on to ending the assigned variable, end the while loop, then terminate the program:

```
ENOUGH TALK  
CHILL  
YOU HAVE BEEN TERMINATED
```

In the end, you should have this:

```
IT'S SHOWTIME  
HEY CHRISTMAS TREE isLessThan20  
YOU SET US UP @NO PROBLEM0  
HEY CHRISTMAS TREE n  
YOU SET US UP 0  
STICK AROUND isLessThan20  
GET TO THE CHOPPER n  
HERE IS MY INVITATION n  
GET UP 1  
ENOUGH TALK  
TALK TO THE HAND n  
GET TO THE CHOPPER isLessThan20  
HERE IS MY INVITATION 20  
LET OFF SOME STEAM BENNET n  
ENOUGH TALK  
CHILL  
YOU HAVE BEEN TERMINATED
```

Now you just need to set the jar package up to run: `/arnoldc$ java -jar ArnoldC.jar count.arnoldc`

Then run your code:

```
/arnoldc$ java count  
1  
2  
3  
4  
5  
  
6  
7
```

```
8
9
10
11
12
13
14
15
16
17
18
19
20
```

If you attempted this tutorial, congratulations again! You now have a small counter.

## Afterthoughts

This just-for-fun open source language is great for general hilarity, but it helps if you know a small amount of Java-based languages. I don't, so it took a bit more time for me to figure out how to use the language. At least I learned something while having fun! I hope you enjoy experimenting with ArnoldC and making something that's amusing to you.



*Tech nomad, working in about anything I can find. Evangelist of silo prevention in the IT space, the importance of information sharing with all teams. Believer in educating all and open source development. Lover of all things tech.*

*All about K8s, chaos and anything new and shiny I can find!*

# Monitor your home's temperature and humidity with Raspberry Pi and Prometheus

By Chris Collins

*Data is beautiful.* As a #CitizenScientist, I enjoy gathering data and trying to make sense of the world around me. At work, we use [Prometheus](#) to gather metric data from our clusters, and at home, I use Prometheus to gather data from my hobbies. This article explores how to take an application—a Python script that gathers temperature and humidity data from a sensor—and instrument it to provide data in a [model that Prometheus can gather](#). I'll also create a [systemd service](#) to start and manage the application.

## What is Prometheus?

Prometheus is an open source monitoring and alerting system that gathers metrics and provides a powerful query language for exploring data. I've written about [setting up Prometheus locally](#) at home. Prometheus is frequently used to gather data from container orchestration clusters such as [Kubernetes](#) and [OpenShift](#).

In my job as a site reliability engineer running OpenShift Dedicated clusters for Red Hat, Prometheus is the core of a robust monitoring and alerting system for all of our clusters, [operators](#), and applications. It is used at huge scale by large enterprise organizations, but it is equally at home, well, at home, collecting data for hobbyist projects.

In my previous article about [using a Raspberry Pi Zero and DHT22 to collect temperature and humidity data](#), I showed how to write a Python script to gather the data and print it to the screen. That is good for checking the data manually for each moment, but it would be far more useful for me to gather and store the data to examine it historically. This is where

Prometheus shines as a time-series database with its own query language and graph capabilities.

## Instrument the app for Prometheus

In a nutshell, instrumenting the application for Prometheus requires taking the data from the sensor, labeling it, and serving it as text over HTTP so that Prometheus can find and store the data. Prometheus will check these text pages, or "targets," at a specified interval, looking for updates to the data. So, the application will need to update the target metrics as new sensor data is received.

The format of the data exposed for Prometheus to gather consists of a key (a metric name—that which is being measured) and a value separated by a space:

```
dht22_temperature{scale="fahrenheit"} 84.01999931335449
```

Prometheus also supports optional labels to make it easier to filter and aggregate data. This application uses labels to differentiate between Celsius and Fahrenheit scales for the `dht22_temperature` metric. The `{scale="fahrenheit"}` is the label in the example above. Check out the [Prometheus data model](#) for more details.

You can modify the script manually to set up a web server and print the sensor data, but Prometheus provides a [Prometheus Python client](#) that makes the process considerably easier. You can install the client using the pip Python package manager. If you don't already have it, install pip using your distribution's package manager, and then use it to install `prometheus-client`. I'm running Raspberry Pi OS on my sensor system, so I'll use `apt-get`, but substitute your package manager of choice:

```
# Install pip
sudo apt-get install pip3
# Install prometheus-client
sudo pip3 install prometheus-client
```

Next, you need to tweak the previous article's sensor script to store the sensor data as [Prometheus gauges](#). A gauge is "a metric that represents a single numerical value that can arbitrarily go up and down," as opposed to, say, a counter, which only goes up. Prometheus has many different metric types. A gauge is perfect to represent temperature and humidity data.

Gauges are provided by the `prometheus_client.Gauge` module, so you need to import the module before you can use gauges in the script. Because `start_http_server` is used later to display the metrics via HTTP, import that now as well:

```
# Import Gauge and start_http_server from prometheus_client
from prometheus_client import Gauge, start_http_server
```

Next, create the gauges to store the humidity and temperature data. The `['scale']` bit adds the "scale" label for the temperature gauge. Then the gauge is initialized with both `celsius` and `fahrenheit` values for the label:

```
# Create Prometheus gauges for humidity and temperature in
# Celsius and Fahrenheit
gh = Gauge('dht22_humidity_percent',
           'Humidity percentage measured by the DHT22 Sensor')
gt = Gauge('dht22_temperature',
           'Temperature measured by the DHT22 Sensor', ['scale'])
# Initialize the labels for the temperature scale
gt.labels('celsius')
gt.labels('fahrenheit')
```

You can set the gauges with the sensor data when checking the sensor:

```
try:
    # Get the sensor data
    humidity, temperature = Adafruit_DHT.read_retry(SENSOR, SENSOR_PIN)
except RuntimeError as e:
    log.error("RuntimeError: {}".format(e))
if humidity is not None and temperature is not None:
    # Update the gauge with the sensor data
    gh.set(humidity)
    gt.labels('celsius').set(temperature)
    gt.labels('fahrenheit').set(celsius_to_fahrenheit(temperature))
```

This is done inside a `while True:` loop (not shown above; see the full script below) to continually update the gauges with data from the sensor.

Finally, Prometheus' `start_metrics_server` will serve the collected metrics via HTTP. This is called before the `while` loop so that the server starts first:

```
# Start the Prometheus metrics server to display the metrics data
metrics_port = 8000
start_http_server(metrics_port)
```

With all this together, the script should look something like this:

```
#!/usr/bin/env python3
import logging
import time
import Adafruit_DHT
from prometheus_client import Gauge, start_http_server
from systemd.journal import JournalHandler
# Setup logging to the Systemd Journal
log = logging.getLogger('dht22_sensor')
log.addHandler(JournalHandler())
log.setLevel(logging.INFO)
# Initialize the DHT22 sensor
# Read data from GPIO4 pin on the Raspberry Pi
SENSOR = Adafruit_DHT.DHT22
SENSOR_PIN = 4
# The time in seconds between sensor reads
READ_INTERVAL = 30.0
# Create Prometheus gauges for humidity and temperature in
# Celsius and Fahrenheit
gh = Gauge('dht22_humidity_percent',
           'Humidity percentage measured by the DHT22 Sensor')
gt = Gauge('dht22_temperature',
           'Temperature measured by the DHT22 Sensor', ['scale'])
# Initialize the labels for the temperature scale
gt.labels('celsius')
gt.labels('fahrenheit')
def celsius_to_fahrenheit(degrees_celsius):
    return (degrees_celsius * 9/5) + 32
def read_sensor():
    try:
        humidity, temperature = Adafruit_DHT.read_retry(SENSOR, SENSOR_PIN)
    except RuntimeError as e:
        # GPIO access may require sudo permissions
        # Other RuntimeError exceptions may occur, but
        # are common. Just try again.
        log.error("RuntimeError: {}".format(e))
    if humidity is not None and temperature is not None:
        gh.set(humidity)
        gt.labels('celsius').set(temperature)
        gt.labels('fahrenheit').set(celsius_to_fahrenheit(temperature))
        log.info("Temp:{0:0.1f}*C, Humidity: {1:0.1f}%".format(temperature,
        humidity))
        time.sleep(READ_INTERVAL)
if __name__ == "__main__":
    # Expose metrics
    metrics_port = 8000
    start_http_server(metrics_port)
```

```
print("Serving sensor metrics on :{}".format(metrics_port))
log.info("Serving sensor metrics on :{}".format(metrics_port))
while True:
    read_sensor()
```

## Set up the systemd unit and logging

The script is ready to go and would work with Prometheus as it is. But I'm running this on headless (i.e., no monitor, keyboard, etc.) Raspberry Pi Zero Ws installed in project boxes with DHT22 sensors, set up in different rooms of the house. I'll add a systemd service to automatically start the script at boot and make sure it keeps running. I'll also take advantage of the systemd journal and send log data from the script (e.g., startup or error messages) to the journal.

The systemd service will be a separate file systemd uses, but it needs the `python3-systemd` package to send logs to the journal from the script. You can install it with `apt-get` (or your package manager):

```
# Install the python3-systemd package for Journal integration
sudo apt-get install python3-systemd
```

You can configure the Python logger within the service monitor script to send logs to the journal by using the `systemd.journal.JournalHandler` module. After importing it, you can add the `JournalHandler` as a handler for the logger:

```
from systemd.journal import JournalHandler
# Setup logging to the Systemd Journal
log = logging.getLogger('dht22_sensor')
log.addHandler(JournalHandler())
log.setLevel(logging.INFO)
```

Now it can log to the journal with `log.info()`. For example:

```
# This will send the message to the Systemd Journal,
# and show up in `systemctl status` and with `journalctl`
log.info("Serving sensor metrics on :{}".format(metrics_port))
```

With the script updated to log to the systemd journal, create a systemd service for the `sensor-metrics.py` script:

```
# /etc/systemd/system/sensor-metrics.service
```

```
[Unit]
Description=DHT22 Sensor Metrics Service
After=network.target
StartLimitIntervalSec=0
[Service]
Type=simple
Restart=always
ExecStart=python3 /opt/sensor-metrics/sensor-metrics.py
[Install]
WantedBy=multi-user.target
```

This merely tells systemd to look for a script in `/opt/sensor-metrics/sensor-metrics.py`, start it, and keep it running. This will become the `sensor-metrics` service.

Link (or move, if you prefer) the `sensor-metrics.py` script to `/opt/sensor-metrics/sensor-metrics.py`:

```
# Create /opt/sensor-metrics and link the sensor-metrics.py script from the
current directory into it
sudo mkdir /opt/sensor-metrics
sudo ln -s $(pwd)/sensor-metrics.py /opt/sensor-metrics/
```

Link the `sensor-metrics.service` file to `/etc/systemd/system`:

```
# Link the sensor-metrics.service file into the Systemd directory
sudo ln -s $(pwd)/sensor-metrics.service /etc/systemd/system/
```

Now you can enable the `sensor-metrics` service to start on boot and check the status:

```
# Enable and start the sensor-metrics.service
sudo systemctl enable sensor-metrics.service
sudo systemctl start sensor-metrics.service
```

Now the service is running and set to start on boot.

To find out if everything is running, check the service status with `systemctl`:

```
sudo systemctl status sensor-metrics.service
```

You should see something like this (if everything is working):

```
• sensor-metrics.service - DHT22 Sensor Metrics Service
  Loaded: loaded (/home/chris/sensor-metrics.service; enabled; vendor preset:
  enabled)
```



```
Active: active (running) since Wed 2021-06-30 03:33:02 BST; 8s ago
Main PID: 4129 (python3)
Tasks: 2 (limit: 877)
CGroup: /system.slice/sensor-metrics.service
└─4129 /usr/bin/python3 /opt/sensor-metrics/sensor-metrics.py
Jun 30 03:33:02 cumulo systemd[1]: Started DHT22 Sensor Metrics Service.
Jun 30 03:33:05 cumulo /opt/sensor-metrics/sensor-metrics.py[4129]: Serving
sensor metrics on :8000
Jun 30 03:33:05 cumulo /opt/sensor-metrics/sensor-metrics.py[4129]: Temp:30.6*C,
Humidity: 47.1%
```

Success!

## Check the metrics target

With the service running and the script modified to collect sensor data in gauges and display it for Prometheus, you can view the data as Prometheus will.

In a browser, navigate to `http://<IP OF YOUR HOST>:8000`, substituting the IP address of the machine running the sensor-metrics service.

You should see a page with several metrics about the Python client (bonus!), as well as the `dht22_temperature` and `dht22_humidity` metrics. It should look something like this:

```
# HELP process_cpu_seconds_total Total user and system CPU time spend in seconds.
# TYPE process_cpu_seconds_total counter
procss_cpu_seconds_total 2.219999999999998
# HELP process_open_fds Number of open file descriptors.
$ TYPE process_open_fds gauge
process_open_fds 8.0
# HELP dht22_humidity_percent Humidity percentage measured by DHT22 Sensor
# TYPE dht22_humidity_percent gauge
dht22_humidity percent 51.5
dht22_temperature Temperature measured by the DHT22 Sensor
dht22_temperature{scale="celcius"} 28.8999999618530273
dht22_temperature{scale="fahrenheit"} 84.01999931335449
```

Data really IS beautiful! Look at that! Humidity *and* temperature in two different scales!

The data updates each time the service script checks the sensor data. Now for the last step: showing Prometheus where to look for all this data.

## Create a Prometheus scrape config

Before moving on, I recommend you read my earlier article about [running Prometheus at home](#) and have an instance already set up. If you haven't, go ahead and do that now (and tell all your friends what a great experience it was). The article shows how to set up Prometheus to read dynamic scrape configs from a file and reload automatically. These scrape configs point Prometheus to the metric data targets it should ingest (i.e., "scrape").

Add a scrape config for the sensor-metrics data to the array (if any) in the scrape config JSON setup in the previous article. Note the array below has the single Prometheus sensor-metrics target. Yours may already have other targets. Just add to the list.

```
// This scrape config target points to the IP Address of the Raspberry Pi and the
// Port used in the sensor-metrics.py script
// Substitute your own IP and the port you chose
[
  {"labels": {"job": "dht22"}, "targets": ["192.168.1.119:8000"]}
]
```

After restarting Prometheus (or waiting for it to find and load the new scrape config), Prometheus will begin looking for metrics data at the target you specified above.

## Bringing it all together

Finally, everything is running, and you can look at the data you're collecting! Open the Prometheus web interface for your Prometheus server. Mine is <http://localhost:9090/graph>, and yours may be the same if you followed along with the previous article. Click on the "graph" tab, and search for `dht22_temperature{scale=~"fahrenheit"}` to see the temperature data being collected.



(Chris Collins, [CC BY-SA 4.0](#))

Well. That's disappointing.

OK, two things:

1. Time-series data is underwhelming at first because you don't have much data yet. It gets better over time.
2. Ambient temperature data takes a somewhat longer period to display anything interesting because it doesn't change much.

So, give it time. Eventually, it'll look much more interesting and get better:



(Chris Collins, [CC BY-SA 4.0](#))

MUCH better!

## Do some #CitizenScience

Prometheus works very well for collecting metric data, storing it as time-series data, and providing a way to explore it. I hope this article has inspired you to perform some #CitizenScience at home, create your own data, and explore!



Chris Collins is an SRE at Red Hat and an OpenSource.com Correspondent with a passion for automation, container orchestration and the ecosystems around them, and likes to recreate enterprise-grade technology at home for fun.

Prior to working at Red Hat, he spent thirteen years with Duke University, variously as a Linux systems administrator, web hosting architecture and team lead, and an automation engineer. In his free time, Chris enjoys Dwarf Fortress, brewing beer,

woodworking, and being a general-purpose geek.

# How to update a Linux symlink

By Alan Formy-Duval

UNIX and Linux users find many uses for links, particularly symbolic links. One way that I like to use symbolic links is to manage configuration backups of various IT equipment.

I have a directory structure to hold everything related to documentation, updates, and other files for the computers and devices on my network. Devices can include routers, access points, NAS servers, and laptops, often of different brands and versions. The configuration backups themselves might be deep within the directory tree, e.g.

`/home/alan/Documents/network/device/NetgearRL5000/config.`

To simplify the backup process, I have a directory in my home called **Configuration**. I use symbolic links from this directory to point to the specific device directory:

```
:~/Configuration/ $ ls -F1
Router@
Accesspoint@
NAS@
```

**Note:** The `-F` option of the `ls` command appends special characters to each file name to represent its type. As shown above, the `@` symbol indicates that these are links.

## Creating a link

The symbolic link **Router** points to the `config` directory of my Netgear RL5000. The command to create it is `ln -s`:

```
$ ln -s /home/alan/Documents/network/device/NetgearRL5000/config Router
```

Then, take a look and confirm with `ls -l`:

```
:~/Configuration/ $ ls -l
```

```
Router -> /home/alan/Documents/network/device/NetgearRL5000/config
NAS -> /home/alan/Documents/network/device/NFSBox/config
...
```

The advantage is that when performing maintenance on this device, I simply browse to `~/Configuration/Router`.

The second advantage of using a symbolic link becomes evident if I decide to replace this router with a new model. I might re-task the old router to be an access point. Therefore, its directory does not get deleted. Instead, I have a new directory that corresponds to the new router, perhaps an ASUS DF-3760. I create the directory and confirm its existence:

```
$ mkdir -p ~/Documents/network/device/ASUSDF-3760/config

:~/Documents/network/device/ $ ls
NetgearRL5000
ASUSDF-3760
NFSBox
...
```

Another example could be if you have several access points throughout your offices. You can use symbolic links to represent each one logically with either a generic name, such as `ap1`, `ap2`, and so on, or you can use descriptive words such as `ap_floor2`, `ap_floor3`, etc. This way, as the physical devices change over time, you do not have to continuously update any processes that might be managing them as they are addressing the links rather than the actual device directories.

## Updating a link

Since my main router has changed, I want the router's symbolic link to point to its directory. I could use the `rm` and `ln` commands to remove and create a new symbolic link, but there is a way to do this in one step using only the `ln` command with a few options:

```
:~/Configuration/ $ ln -vfns ~/Documents/network/device/ASUSDF-3760/config/
Router
'Router' -> '/home/alan/Documents/network/device/ASUSDF-3760/config/'
:~/Configuration/ $ ls -l
Router -> /home/alan/Documents/network/device/ASUSDF-3760/config
NAS -> /home/alan/Documents/network/device/NFSBox/config
```

The options, according to the man page, are as follow:

- **-v, --verbose** Print name of each linked file
- **-f, --force** Remove destination file (necessary since a link already exists)
- **-n, --no-dereference** Treat LINK\_NAME as a normal file if it is a symbolic link to a directory
- **-s, --symbolic** Make symbolic links instead of hard links

## Wrap up

Links are one of the most powerful features of UNIX and Linux file systems. Other operating systems have tried to mimic this capability, but those never worked as well or were as usable due to the lack of a fundamental link design in their file systems.

The demonstration above is only one possibility of many to take advantage of links for seamlessly navigating an ever-changing directory structure in a living production environment. Links provides the flexibility needed in an organization that is never static for long.



Alan has 20 years of IT experience, mostly in the Government and Financial sectors. He started as a Value Added Reseller before moving into Systems Engineering. Alan's background is in high-availability clustered apps. He wrote the 'Users and Groups' and 'Apache and the Web Stack' chapters in the Oracle Press/McGraw Hill 'Oracle Solaris 11 System Administration' book. He earned his Master of Science in Information Systems from George Mason University.

# 5 common bugs in C programming and how to fix them

By Jim Hall

Even the best programmers can create programming bugs. Depending on what your program does, these bugs could introduce security vulnerabilities, cause the program to crash, or create unexpected behavior.

The C programming language sometimes gets a bad reputation because it is not memory safe like more recent programming languages, including Rust. But with a little extra code, you can avoid the most common and most serious C programming bugs. Here are five bugs that can break your application and how you can avoid them:

## 1. Uninitialized variables

When the program starts up, the system will assign it a block of memory that the program uses to store data. That means your variables will get whatever random value was in memory when the program started.

Some environments will intentionally "zero out" the memory as the program starts up, so every variable starts with a zero value. And it can be tempting to assume in your programs that all variables will begin at zero. However, the C programming specification says that the system does not initialize variables.

Consider a sample program that uses a few variables and two arrays:

```
#include <stdio.h>
#include <stdlib.h>
int
main()
{
    int i, j, k;
```



```

int numbers[5];
int *array;
puts("These variables are not initialized:");
printf("  i = %d\n", i);
printf("  j = %d\n", j);
printf("  k = %d\n", k);
puts("This array is not initialized:");
for (i = 0; i < 5; i++) {
    printf("  numbers[%d] = %d\n", i, numbers[i]);
}
puts("malloc an array ...");
array = malloc(sizeof(int) * 5);
if (array) {
    puts("This malloc'ed array is not initialized:");
    for (i = 0; i < 5; i++) {
        printf("  array[%d] = %d\n", i, array[i]);
    }
    free(array);
}
/* done */
puts("Ok");
return 0;
}

```

The program does not initialize the variables, so they start with whatever values the system had in memory at the time. Compiling and running this program on my Linux system, you'll see that some variables happen to have "zero" values, but others do not:

```

These variables are not initialized:
  i = 0
  j = 0
  k = 32766
This array is not initialized:
  numbers[0] = 0
  numbers[1] = 0
  numbers[2] = 4199024
  numbers[3] = 0
  numbers[4] = 0
malloc an array ...
This malloc'ed array is not initialized:
  array[0] = 0
  array[1] = 0
  array[2] = 0
  array[3] = 0
  array[4] = 0
Ok

```

Fortunately, the `i` and `j` variables start at zero, but `k` has a starting value of 32766. In the numbers array, most elements also happen to start with zero, except the third element, which gets an initial value of 4199024.

Compiling the same program on a different system further shows the danger in uninitialized variables. Don't assume "all the world runs Linux" because one day, your program might run on a different platform. For example, here's the same program running on FreeDOS:

```
These variables are not initialized:
i = 0
j = 1074
k = 3120
This array is not initialized:
numbers[0] = 3106
numbers[1] = 1224
numbers[2] = 784
numbers[3] = 2926
numbers[4] = 1224
malloc an array ...
This malloc'ed array is not initialized:
array[0] = 3136
array[1] = 3136
array[2] = 14499
array[3] = -5886
array[4] = 219
Ok
```

Always initialize your program's variables. If you assume a variable will start with a zero value, add the extra code to assign zero to the variable. This extra bit of typing upfront will save you headaches and debugging later on.

## 2. Going outside of array bounds

In C, arrays start at array index zero. That means an array that is ten elements long goes from 0 to 9, or an array that is a thousand elements long goes from 0 to 999.

Some programmers sometimes forget this and introduce "off by one" bugs where they reference the array starting at one. In an array that is five elements long, the value the programmer intended to find at array element "5" is not actually the fifth element of the array. Instead, it is some other value in memory, not associated with the array at all.

Here's an example that goes well outside the array bounds. The program starts with an array that's only five elements long but references array elements from outside that range:

```

#include <stdio.h>
#include <stdlib.h>
int
main()
{
    int i;
    int numbers[5];
    int *array;
    /* test 1 */
    puts("This array has five elements (0 to 4)");
    /* initialize the array */
    for (i = 0; i < 5; i++) {
        numbers[i] = i;
    }
    /* oops, this goes beyond the array bounds: */
    for (i = 0; i < 10; i++) {
        printf(" numbers[%d] = %d\n", i, numbers[i]);
    }
    /* test 2 */
    puts("malloc an array ...");
    array = malloc(sizeof(int) * 5);
    if (array) {
        puts("This malloc'ed array also has five elements (0 to 4)");
        /* initialize the array */
        for (i = 0; i < 5; i++) {
            array[i] = i;
        }
        /* oops, this goes beyond the array bounds: */
        for (i = 0; i < 10; i++) {
            printf(" array[%d] = %d\n", i, array[i]);
        }
        free(array);
    }
    /* done */
    puts("Ok");
    return 0;
}

```

Note that the program initializes all the values of the array, from 0 to 4, but then tries to read 0 to 9 instead of 0 to 4. The first five values are correct, but after that you don't know what the values will be:

```

This array has five elements (0 to 4)
numbers[0] = 0
numbers[1] = 1
numbers[2] = 2
numbers[3] = 3

```

```
numbers[4] = 4
numbers[5] = 0
numbers[6] = 4198512
numbers[7] = 0
numbers[8] = 1326609712
numbers[9] = 32764
malloc an array ...
This malloc'ed array also has five elements (0 to 4)
array[0] = 0
array[1] = 1
array[2] = 2
array[3] = 3
array[4] = 4
array[5] = 0
array[6] = 133441
array[7] = 0
array[8] = 0
array[9] = 0
Ok
```

When referencing arrays, always keep track of its size. Store that in a variable; don't hard-code an array size. Otherwise, your program might stray outside the array bounds when you later update it to use a different array size, but you forget to change the hard-coded array length.

### 3. Overflowing a string

Strings are just arrays of a different kind. In the C programming language, a string is an array of `char` values, with a zero character to indicate the end of the string.

And so, like arrays, you need to avoid going outside the range of the string. This is sometimes called *overflowing a string*.

One easy way to overflow a string is to read data with the `gets` function. The `gets` function is very dangerous because it doesn't know how much data it can store in a string, and it naively reads data from the user. This is fine if your user enters short strings like `foo` but can be disastrous when the user enters a value that is too long for your string value.

Here's a sample program that reads a city name using the `gets` function. In this program, I've also added a few unused variables to show how string overflow can affect other data:

```
#include <stdio.h>
#include <string.h>
int
main()
```

```

{
    char name[10];                /* Such as "Chicago" */
    int var1 = 1, var2 = 2;
    /* show initial values */
    printf("var1 = %d; var2 = %d\n", var1, var2);
    /* this is bad .. please don't use gets */
    puts("Where do you live?");
    gets(name);
    /* show ending values */
    printf("<%s> is length %d\n", name, strlen(name));
    printf("var1 = %d; var2 = %d\n", var1, var2);
    /* done */
    puts("Ok");
    return 0;
}

```

That program works fine when you test for similarly short city names, like **Chicago** in Illinois or **Raleigh** in North Carolina:

```

var1 = 1; var2 = 2
Where do you live?
Raleigh
<Raleigh> is length 7
var1 = 1; var2 = 2
Ok

```

The Welsh town of

**Llanfairpwllgwyngyllgogerychwyrndrobwlllandysiliogogogoch** has one of the longest names in the world. At 58 characters, this string goes well beyond the 10 characters reserved in the `name` variable. As a result, the program stores values in other areas of memory, including the values of `var1` and `var2`:

```

var1 = 1; var2 = 2
Where do you live?
Llanfairpwllgwyngyllgogerychwyrndrobwlllandysiliogogogoch
<Llanfairpwllgwyngyllgogerychwyrndrobwlllandysiliogogogoch> is length 58
var1 = 2036821625; var2 = 2003266668
Ok
Segmentation fault (core dumped)

```

Before aborting, the program used the long string to overwrite other parts of memory. Note that `var1` and `var2` no longer have their starting values of 1 and 2.

Avoid `gets`, and use safer methods to read user data. For example, the `getline` function will allocate enough memory to store user input, so the user cannot accidentally overflow the string by entering a long value.

## 4. Freeing memory twice

One of the rules of good C programming is, "if you allocate memory, you should free it." Programs can allocate memory for arrays and strings using the `malloc` function, which reserves a block of memory and returns a pointer to the starting address in memory. Later, the program can release the memory using the `free` function, which uses the pointer to mark the memory as unused.

However, you should only use the `free` function once. Calling `free` a second time will result in unexpected behavior that will probably break your program. Here's a short example program to show that. It allocates memory, then immediately releases it. But like a forgetful-but-methodical programmer, I also freed the memory at the end of the program, resulting in freeing the same memory twice:

```
#include <stdio.h>
#include <stdlib.h>
int
main()
{
    int *array;
    puts("malloc an array ...");
    array = malloc(sizeof(int) * 5);
    if (array) {
        puts("malloc succeeded");
        puts("Free the array...");
        free(array);
    }
    puts("Free the array...");
    free(array);
    puts("Ok");
}
```

Running this program causes a dramatic failure on the second use of the `free` function:

```
malloc an array ...
malloc succeeded
Free the array...
Free the array...
free(): double free detected in tcache 2
```

Aborted (core dumped)

Avoid calling `free` more than once on an array or string. One way to avoid freeing memory twice is to locate the `malloc` and `free` functions in the same function.

For example, a solitaire program might allocate memory for a deck of cards in the main function, then use that deck in other functions to play the game. Free the memory in the main function, rather than some other function. Keeping the `malloc` and `free` statements together helps to avoid freeing memory more than once.

## 5. Using invalid file pointers

Files are a handy way to store data. For example, you might store configuration data for your program in a file called `config.dat`. The Bash shell reads its initial script from `.bash_profile` in the user's home directory. The GNU Emacs editor looks for the file `.emacs` for its starting values. And the Zoom meeting client uses the `zoomus.conf` file to read its program configuration.

So the ability to read data from a file is important for pretty much all programs. But what if the file you want to read isn't there?

To read a file in C, you first open the file using the `fopen` function, which returns a stream pointer to the file. You can use this pointer with other functions to read data, such as `fgetc` to read the file one character at a time.

If the file you want to read isn't there or isn't readable by your program, then the `fopen` function will return `NULL` as the file pointer, which is an indication the file pointer is invalid. But here's a sample program that innocently does not check if `fopen` returned `NULL` and tries to read the file regardless:

```
#include <stdio.h>
int
main()
{
    FILE *pfile;
    int ch;
    puts("Open the FILE.TXT file ...");
    pfile = fopen("FILE.TXT", "r");
    /* you should check if the file pointer is valid, but we skipped that */
    puts("Now display the contents of FILE.TXT ...");
    while ((ch = fgetc(pfile)) != EOF) {
        printf("<%c>", ch);
    }
}
```

```
}  
fclose(pfile);  
/* done */  
puts("Ok");  
return 0;  
}
```

When you run this program, the first call to `fgetc` results in a spectacular failure, and the program immediately aborts:

```
Open the FILE.TXT file ...  
Now display the contents of FILE.TXT ...  
Segmentation fault (core dumped)
```

Always check the file pointer to ensure it's valid. For example, after calling `fopen` to open a file, check the pointer's value with something like `if (pfile != NULL)` to ensure that the pointer is something you can use.

We all make mistakes, and programming bugs happen to the best of programmers. But if you follow these guidelines and add a little extra code to check for these five types of bugs, you can avoid the most serious C programming mistakes. A few lines of code up front to catch these errors may save you hours of debugging later.



Jim Hall is an open source software advocate and developer, best known for usability testing in GNOME and as the founder + project coordinator of FreeDOS. At work, Jim is CEO of Hallmentum, an IT executive consulting company that provides hands-on IT Leadership training, workshops, and coaching.



# Calculate date and time ranges in Groovy

By Chris Hermansen

Every so often, I need to do some calculations related to dates. A few days ago, a colleague asked me to set up a new project definition in our (open source, of course!) project management system. This project is to start on the 1st of August and finish on the 31st of December. The service to be provided is budgeted at 10 hours per week.

So, yeah, I had to figure out how many weeks between 2021-08-01 and 2021-12-31 inclusive.

This is the perfect sort of problem to solve with a tiny [Groovy](#) script.

## Install Groovy on Linux

Groovy is based on Java, so it requires a Java installation. Both a recent and decent version of Java and Groovy might be in your Linux distribution's repositories. Alternately, you can install Groovy by following the instructions on the [groovy-lang.org](http://groovy-lang.org).

A nice alternative for Linux users is [SDKMan](#), which can be used to get multiple versions of Java, Groovy, and many other related tools. For this article, I'm using my distro's OpenJDK11 release and SDKMan's latest Groovy release.

## Solving the problem with Groovy

Since Java 8, time and date calculations have been folded into a new package called **java.time**, and Groovy provides access to that. Here's the script:

```
import java.time.*
import java.time.temporal.*
def start = LocalDate.parse('2021-08-01', 'yyyy-MM-dd')
```

```
def end = LocalDate.parse('2022-01-01', 'yyyy-MM-dd')
println "${ChronoUnit.WEEKS.between(start,end)} weeks between $start and $end"
```

Copy this code into a file called **wb.groovy** and run it on the command line to see the results:

```
$ groovy wb.groovy
21 weeks between 2021-08-01 and 2022-01-01
```

Let's review what's going on.

## Date and time

The [java.time.LocalDate](#) class provides many useful static methods (like **parse()** shown above, which lets us convert from a string to a **LocalDate** instance according to a pattern, in this case, **'yyyy-MM-dd'**). The format characters are explained in quite a number of places—for example, the documentation for [java.time.format.DateTimeFormat](#). Notice that **M** represents "month," not **m**, which represents "minute." So this pattern defines a date formatted as a four-digit year, followed by a hyphen, followed by a two-digit month number (1-12), followed by another hyphen, followed by a two-digit day-of-month number (1-31).

Notice as well that in Java, **parse()** requires an instance of **DateTimeFormat**:

```
parse(CharSequence text, DateTimeFormatter formatter)
```

As a result, parsing becomes a two-step operation, whereas Groovy provides an additional version of **parse()** that accepts the format string directly in place of the **DateTimeFormat** instance.

The [java.time.temporal.ChronoUnit](#) class, actually an **Enum**, provides several **Enum constants**, like **WEEKS** (or **DAYS**, or **CENTURIES...**) which in turn provide the **between()** method that allows us to calculate the interval of those units between two **LocalDates** (or other similar date or time data types). Note that I used January 1, 2022, as the value for **end**; this is because **between()** spans the time period starting on the first date given up to but not including the second date given.

## More date arithmetic

Every so often, I need to know how many working days are in a specific time frame (like, say, a month). This handy script will calculate that for me:

```

import java.time.*
def holidaySet = [LocalDate.parse('2021-01-01'), LocalDate.parse('2021-04-02'),
    LocalDate.parse('2021-04-03'), LocalDate.parse('2021-05-01'),
    LocalDate.parse('2021-05-15'), LocalDate.parse('2021-05-16'),
    LocalDate.parse('2021-05-21'), LocalDate.parse('2021-06-13'),
    LocalDate.parse('2021-06-21'), LocalDate.parse('2021-06-28'),
    LocalDate.parse('2021-06-16'), LocalDate.parse('2021-06-18'),
    LocalDate.parse('2021-08-15'), LocalDate.parse('2021-09-17'),
    LocalDate.parse('2021-09-18'), LocalDate.parse('2021-09-19'),
    LocalDate.parse('2021-10-11'), LocalDate.parse('2021-10-31'),
    LocalDate.parse('2021-11-01'), LocalDate.parse('2021-11-21'),
    LocalDate.parse('2021-12-08'), LocalDate.parse('2021-12-19'),
    LocalDate.parse('2021-12-25')] as Set
def weekendDaySet = [DayOfWeek.SATURDAY, DayOfWeek.SUNDAY] as Set
int calcWorkingDays(start, end, holidaySet, weekendDaySet) {
    (start..<end).inject(0) { subtotal, d ->
        if (!(d in holidaySet || DayOfWeek.from(d) in weekendDaySet))
            subtotal + 1
        else
            subtotal
    }
}
def start = LocalDate.parse('2021-08-01')
def end = LocalDate.parse('2021-09-01')
println "${calcWorkingDays(start,end,holidaySet,weekendDaySet)} working day(s)
between $start and $end"

```

Copy this code into a file called **wdb.groovy** and run it from the command line to see the results:

```

$ groovy wdb.groovy
22 working day(s) between 2021-08-01 and 2021-09-01

```

Let's review this.

First, I create a set of holiday dates (these are Chile's "días feriados" for 2021, in case you wondered) called holidaySet. Note that the default pattern for **LocalDate.parse()** is 'yyyy-MM-dd', so I've left the pattern out here. Note as well that I'm using the Groovy shorthand **[a,b,c]** to create a **List** and then coercing it to a **Set**.

Next, I want to skip Saturdays and Sundays, so I create another set incorporating two **enum** values of **java.time.DayOfWeek**—**SATURDAY** and **SUNDAY**.

Then I define a method **calcWorkingDays()** that takes as arguments the start date, the end date (which following the previous example of **between()** is the first value outside the range I want to consider), the holiday set, and the weekend day set. Line by line, this method:

- Defines a range between **start** and **end**, open on the **end**, (that's what **<end** means) and executes the closure argument passed to the **inject()** method (**inject()** implements the 'reduce' operation on **List** in Groovy) on the successive elements **d** in the range:
  - As long as **d** is neither in the **holidaySet** nor in the **weekendDaySet**, increments the **subtotal** by 1
- Returns the value of the result returned by **inject()**

Next, I define the **start** and **end** dates between which I want to calculate working days.

Finally, I call **println** using a Groovy **GString** to evaluate the **calcWorkingDays()** method and display the result.

Note that I could have used the **each** closure instead of **inject**, or even a **for** loop. I could have also used Java Streams rather than Groovy ranges, lists, and closures. Lots of options.

## But why not use groovy.Date?

Some of you old Groovy users may be wondering why I'm not using good old **groovy.Date**. The answer is, I could use it. But Groovy Date is based on Java Date, and there are some good reasons for moving to **java.time**, even though Groovy Date added quite a few nice things to Java Date.

For me, the main reason is that there are some not-so-great design decisions buried in the implementation of Java Date, the worst being that it is unnecessarily mutable. I spent a while tracking down a weird bug that arose from my poor understanding of the **clearTime()** method on Groovy Date. I learned it actually clears the time field of the date instance, rather than returning the date value with the time part set to '00:00:00'.

Date instances also aren't thread-safe, which can be kind of challenging for multithreaded applications.

Finally, having both date and time wrapped up in a single field isn't always convenient and can lead to some weird data modeling contortions. Think, for instance, of a day on which multiple events occur: Ideally, the *date* field would be on the day, and the *time* field would be on each event; but that's not easy to do with Groovy Date.

## Groovy is groovy

Groovy is an Apache project, and it provides a simplified syntax for Java so you can use it for quick and simple scripts in addition to complex applications. You retain the power of Java, but you access it with an efficient toolset. [Try it soon](#), and see if you find your groove with Groovy.

---



*Seldom without a computer of some sort since graduating from the University of British Columbia in 1978, I have been a full-time Linux user since 2005, a full-time Solaris and SunOS user from 1986 through 2005, and UNIX System V user before that.*

*On the technical side of things, I have spent a great deal of my career as a consultant, doing data analysis and visualization; especially spatial data analysis. I have a substantial amount of related programming experience, using C, awk, Java, Python,*

*PostgreSQL, PostGIS and lately Groovy. I'm looking at Julia with great interest. I have also built a few desktop and web-based applications, primarily in Java and lately in Grails with lots of JavaScript on the front end and PostgreSQL as my database of choice.*

*Aside from that, I spend a considerable amount of time writing proposals, technical reports and - of course - stuff on [Opensource.com](#).*

# How a college student founded a free and open source operating system

By Don Watkins and Joshua Allen Holm

[Jim Hall](#) is best known as the computer programmer who founded the FreeDOS project. Jim began the project in 1994 as a replacement for MS-DOS while he was still a student at the University of Wisconsin–River Falls. Jim created FreeDOS in response to Microsoft ending support for MS-DOS in 1994. Recently Jim agreed to an email interview. Correspondent Joshua Allen Holm joined me in posing the following questions to Jim.

## **Don Watkins: What kind of skill set invites you to write your own operating system?**

I think even a beginner can get started writing an operating system like FreeDOS, although it would take a more advanced programmer to write the kernel.

[I am a self-taught programmer.](#) I learned about programming from an early age by tinkering on our Apple II computer at home. Much later, I learned C programming—my brother was a computer science student when I was a physics student, and he introduced me to C. I picked up the rest by reading books and writing my own programs.

I wrote a lot of small utilities that enhanced my command line on MS-DOS or even replaced certain DOS commands. And you can write a lot of those programs even with a basic level of programming experience. You can write file utilities like FIND, FC, CHOICE, TYPE, MORE, or COPY—or user commands like ECHO or CLS—with only an introduction to C programming. With a bit of practice, you can write system-level programs like ATTRIB, or the COMMAND shell.

## **DW: Were you inspired by Linus Torvalds when you decided to write your own version of DOS and how did that contribute to your licensing decision?**

In a way, yes. I really liked DOS and had been using it since the early 1980s. I ran MS-DOS on my personal computer at university. But in 1993, I discovered Linux.

I really liked the Unix systems in our campus computer lab, where I spent much of my time as an undergraduate university student. When I heard about Linux, a free version of Unix that I could run on my '386 computer at home, I immediately wanted to try it out. [My first Linux distribution](#) was Softlanding Linux System (SLS) 1.03, with Linux kernel 0.99 alpha patch level 11. That required a whopping 2MB of RAM, or 4MB if you wanted to compile programs, and 8MB to run X windows.

I dual-booted Linux with MS-DOS. I booted into Linux most of the time, but I still booted back into MS-DOS to write papers in a word processor, to use a spreadsheet program to analyze data for my lab classes or to play my favorite DOS games.

In 1994, I heard that Microsoft planned to “do away” with MS-DOS. The next version of Windows would eliminate DOS. I didn’t like that, and I still wanted to run DOS. I decided that if folks could come together over the Internet to write something like Linux, surely, we could do the same with DOS. After all, DOS was fairly simple compared to Linux.

On June 29, 1994, [I announced the “PD-DOS” project](#) to write our own DOS. I called it “PD” because I thought it would be in the public domain. But I quickly learned about the GNU General Public License that the Linux kernel used, and decided that was a much better license. No one could take our source code and create a proprietary version of DOS. We changed the name to “Free-DOS” after another week or so. We later dropped the hyphen to become “FreeDOS.”

### **Joshua Allen Holm: What are the advantages of using FreeDOS over alternative ways of running DOS applications (e.g., DOSBox)?**

Using DOSBox to run DOS applications in Linux is a great way to run certain DOS applications. But DOSBox is really intended to launch a single DOS program, like a game. The DOS command line is pretty limited in DOSBox.

In contrast, FreeDOS provides a full DOS command line. We include all of the commands you remember from classic DOS, and added other commands and utilities to do new things. FreeDOS also includes compilers and assemblers so developers can write new programs, utilities, and tools to make your DOS experience more useful, Internet programs to help you get on a network, and even open source games.

## **JAH: Looking back over the years you have worked on FreeDOS, is there anything you would have done differently?**

There's only one event that I wish I could take back. I occasionally reach out to companies that sold DOS applications in the 1980s and 1990s and ask them if they will release the source code to their old DOS programs under an open source software license. That's a great way to contribute to the open source community.

One popular DOS program was a replacement for the MS-DOS COMMAND shell. 4DOS, by JP Software, was an extremely powerful DOS shell and included many modern features. For example, 4DOS supported built-in aliases, color-coded directory listings, and a "swapping" mechanism that freed up more conventional memory to run programs.

I contacted JP Software to ask if they would release the source code to 4DOS under an open source software license. JP Software had stopped supporting DOS, and instead focused on a similar replacement for the CMD shell in Windows NT, called 4NT. They were interested in releasing the source code to 4DOS but were concerned that someone might take the 4DOS source code and release a version for Windows. In effect, that would put JP Software in competition with their older product.

I still didn't understand the fine points of open source software licenses, and I gave them bad advice. I suggested they might start with an existing open source license and add a term that said you could only run it on DOS. They then released the 4DOS source code under a modified version of the MIT license.

Unfortunately, limiting where you can run the software violates one of the tenets of open source software and free software. Users should be able to run open source software anywhere, and for any use. An open source license isn't "open source" if you are limited to running it only on one operating system.

So despite best intentions, I gave JP Software really bad advice there, and 4DOS isn't actually open source software. We used to include 4DOS in FreeDOS—but as we are preparing the FreeDOS 1.3 release, we want to be careful to only include open source software. So FreeDOS 1.3 RC4 ("release candidate 4") does not include 4DOS.

## **JAH: What are some interesting ways people are using FreeDOS?**

Over the years, I've seen people use FreeDOS to do a lot of really interesting things!

One of the earliest cool examples was someone who built pinball machines like you used to see in arcades. He embedded a version of FreeDOS to track the score and update the video



screen on the back of the machine. I don't know exactly how he did this, but my guess is every target or bumper on the pinball board probably generated a keyboard event. You can write a DOS application to read the "keyboard" and update the score based on that.

A few years ago, a user found a video of a train control system in Russia [that ran on a FreeDOS PC](#). They rebooted the computer, and if you freeze the video at the right point, you can briefly see the FreeDOS kernel starting up. It disappears quickly, but you can see it at 0:07 in the video.

More recently, I saw someone had managed to boot an original IBM PC 5150 with FreeDOS [from a vinyl record](#), using the 5150's rarely used cassette tape storage port. It's really cool to see FreeDOS being used this way. It's a method that I would never have thought to try, but sometimes you have to do something just for the fun of it.

### **JAH: Why work on DOS in 2021?**

We still work on DOS in 2021 for a few reasons. I guess the first reason is that DOS is still interesting. We've added a lot to FreeDOS over the years. Where the original MS-DOS had a limited set of commands, FreeDOS includes dozens of useful utilities and tools, including editors, compilers, assemblers, games, and other neat programs.

But it has to be more than just a cool hobby. I find that working on FreeDOS makes for a very interesting programming challenge. In modern systems like Linux, you can take advantage of a lot of memory at once, and you can address it all in one big block. As a result, many programmers will load a lot of libraries and other code to create their projects. This is a very easy way to build a complicated project. You can build a very complex system in a very short time this way. And for many systems, time to market is the most important factor.

Loading a bunch of libraries and other code blocks is very inefficient, however. You may have the same basic functionality implemented half a dozen ways across the different libraries because each library implements something their own way. So your code grows and requires more memory.

Maybe that's not a problem on a desktop PC. I run Linux, and my modern desktop PC has 32GB of memory. Loading a bunch of stuff into memory isn't a big deal. But on a shared server, where you might have multiple instances of that project running, you'll quickly run into memory limitations. How many instances can you run at the same time on a server? That 32GB of memory starts to look pretty slim.

You can't load all of that into memory on a DOS machine. To remain compatible with the original DOS, FreeDOS has all the limitations of DOS. When MS-DOS was popular, a powerful PC might have had 4MB, 8MB, or even 16MB of extended memory. But the computer only had 640kb of "main" memory, due to how DOS addressed memory. And that's megabytes and kilobytes, not gigabytes. A kilobyte is a thousand bytes (the basic unit of memory). A megabyte is a thousand kilobytes. And a gigabyte is a thousand megabytes. So today's computers have memory that is about 1,000,000 more than a DOS computer.

By programming on a limited system like FreeDOS, you constantly have to think about the tradeoffs. How much memory does my program really need to do its job? Is it faster to read a file into memory to work on it, or process the file one bit at a time? And you're always keeping in mind what libraries and other code you use in your program. A DOS program can only be so big, so you need to be careful about how you write a DOS program.

When you write DOS programs all the time, you get really good at optimizing a program. You think about programming in a different way, because you're always considering how to do something more efficiently. That's a challenge, but an interesting one.

### **DW: How big is the FreeDOS community?**

FreeDOS was a very popular project throughout the 1990s and into the early 2000s, but the community isn't as big these days. It's hard to estimate the size of the community. I'd say we have a few dozen members who are very active. And we have a few dozen others who reappear occasionally to post new versions of their programs. I think to maintain an active community that's still working on an open source DOS from 1994 is a great sign.

Some members have been with us from the very beginning, and I'm really thankful to count them as friends. We do [video hangouts on a semi-regular basis](#). It's great to finally "meet" the folks I've only exchanged emails with over the years.

It's meetings like this when I remember open source is more than just writing code; it's about a community. And while I've always done well with our virtual community that communicates via email, I really appreciated getting to talk to people without the asynchronous delay or artificial filter of email—making that real-time connection means a lot to me.

### **DW: How does someone get involved in the community?**

I think our community is very welcoming, so anyone is free to join. We communicate via an email list, which you can find on the [FreeDOS website](#). Join the freedos-user email list if you want to talk about FreeDOS or ask for help. Developers should join the freedos-devel email

list; that's where most of the FreeDOS developers hang out. Our email list volume is pretty low, so you aren't likely to fill up your inbox by subscribing to either email list.

A great way to get started is by writing or updating documentation, or by fixing bugs. I think that's true of pretty much every open source project out there. We always need folks to work on the documentation and fix bugs. But for a project like FreeDOS, I think reading through the documentation is important if you're new to DOS. A common mistake for newcomers is thinking of FreeDOS as a stripped-down version of Linux, when in fact DOS uses a different memory and execution model. You can learn about that by reading the documentation, which is why I recommend new contributors start there.

For the more adventurous, we maintain a list of priority projects on our website. If you'd like to contribute to FreeDOS, but aren't sure what needs work, you might consider tackling one or more of these projects:

- If you've got some programming experience:
  - Port FreeDOS utilities to OpenWatcom C and NASM—our preferred C compiler and Assembler for FreeDOS.
  - Port GNU utilities to FreeDOS, such as using IA-16 GCC (while IA-16 GCC requires a '386 or better to compile, programs compiled with IA-16 GCC run on all CPUs)
  - Create a new alternative shell, similar to COMMAND.COM but with expanded BAT programming
  - Add international language support to a FreeDOS program that currently only supports one language.
- If you're a highly-skilled DOS developer:
  - Write a guest tool like VMSMOUNT for VirtualBox
  - Write a driver for modern sound cards
  - Add some kind of UEFI bootstrap BIOS emulator, perhaps implemented as a CSM

We like to make it easy for new contributors to get started in FreeDOS, and we welcome everyone who wants to work on FreeDOS. If you still don't know how to contribute, feel free to ask on the email list.

# 7 summer book recommendations from open source enthusiasts

By Joshua Allen Holm

It is my great pleasure to introduce Opensource.com's 2022 summer reading list. This year's list contains seven wonderful reading recommendations from members of the Opensource.com community. You will find a nice mix of books covering everything from a fun cozy mystery to non-fiction works that explore thought-provoking topics. I hope you find something on this list that interests you.

Enjoy!

**97 Things Every Java Programmer Should Know: Collective Wisdom from the Experts, edited by Kevlin Henney and Trisha Gee**

*Recommendation written by [Seth Kenlon](#)*

Written by 73 different authors working in all aspects of the software industry, the secret to this book's greatness is that it actually applies to much more than just Java programming. Of course, some chapters lean into Java, but there are topics like Be aware of your container surroundings, Deliver better software, faster, and Don't hIDE your tools that apply to development regardless of language.

Better still, some chapters apply to life in general. Break problems and tasks into small chunks is good advice on how to tackle any problem, Build diverse teams is important for every group of collaborators, and From puzzles to products is a fascinating look at how the mind of a puzzle-solver can apply to many different job roles.

Each chapter is just a few pages, and with 97 to choose from, it's easy to skip over the ones that don't apply to you. Whether you write Java code all day, just dabble, or if you haven't yet started, this is a great book for geeks interested in code and the process of software development.

## **A City is Not a Computer: Other Urban Intelligences, by Shannon Mattern**

Recommendation written by [Scott Nesbitt](#)

These days, it's become fashionable (if not inevitable) to make everything *smart*: Our phones, our household appliances, our watches, our cars, and, especially, our cities.

With the latter, that means putting sensors everywhere, collecting data as we go about our business, and pushing information (whether useful or not) to us based on that data.

This begs the question, does embedding all that technology in a city make it smart? In *A City Is Not a Computer*, Shannon Mattern argues that it doesn't.

A goal of making cities smart is to provide better engagement with and services to citizens. Mattern points out that smart cities often "aim to merge the ideologies of technocratic managerialism and public service, to reprogram citizens as 'consumers' and 'users'." That, instead of encouraging citizens to be active participants in their cities' wider life and governance.

Then there's the data that smart systems collect. We don't know what and how much is being gathered. We don't know how it's being used and by whom. There's *so much* data being collected that it overwhelms the municipal workers who deal with it. They can't process it all, so they focus on low-hanging fruit while ignoring deeper and more pressing problems. That definitely wasn't what cities were promised when they were sold smart systems as a balm for their urban woes.

*A City Is Not a Computer* is a short, dense, well-researched polemic against embracing smart cities because technologists believe we should. The book makes us think about the purpose of a smart city, who really benefits from making a city smart, and makes us question whether we need to or even should do that.

## **git sync murder, by Michael Warren Lucas**

Recommendation written by [Joshua Allen Holm](#)

Dale Whitehead would rather stay at home and connect to the world through his computer's terminal, especially after what happened at the last conference he attended. During that conference, Dale found himself in the role of an amateur detective solving a murder. You can read about that case in the first book in this series, *git commit murder*.

Now, back home and attending another conference, Dale again finds himself in the role of detective. *git sync murder* finds Dale attending a local tech conference/sci-fi convention

where a dead body is found. Was it murder or just an accident? Dale, now the "expert" on these matters, finds himself dragged into the situation and takes it upon himself to figure out what happened. To say much more than that would spoil things, so I will just say *git sync murder* is engaging and enjoyable to read. Reading *git commit murder* first is not necessary to enjoy *git sync murder*, but I highly recommend both books in the series.

Michael Warren Lucas's *git murder* series is perfect for techies who also love cozy mysteries. Lucas has literally written the book on many complex technical topics, and it carries over to his fiction writing. The characters in *git sync murder* talk tech at conference booths and conference social events. If you have not been to a conference recently because of COVID and miss the experience, Lucas will transport you to a tech conference with the added twist of a murder mystery to solve. Dale Whitehead is an interesting, if somewhat unorthodox, cozy mystery protagonist, and I think most Opensource.com readers would enjoy attending a tech conference with him as he finds himself thrust into the role of amateur sleuth.

### **Kick Like a Girl, by Melissa Di Donato Roos**

*Recommendation written by [Joshua Allen Holm](#)*

Nobody likes to be excluded, but that is what happens to Francesca when she wants to play football at the local park. The boys won't play with her because she's a girl, so she goes home upset. Her mother consoles her by relating stories about various famous women who have made an impact in some significant way. The historical figures detailed in *Kick Like a Girl* include women from throughout history and from many different fields. Readers will learn about Frida Kahlo, Madeleine Albright, Ada Lovelace, Rosa Parks, Amelia Earhart, Marie Curie, Valentina Tereshkova, Florence Nightingale, and Malala Yousafzai. After hearing the stories of these inspiring figures, Francesca goes back to the park and challenges the boys to a football match.

*Kick Like a Girl* features engaging writing by Melissa Di Donato Roos (SUSE's CEO) and excellent illustrations by Ange Allen. This book is perfect for young readers, who will enjoy the rhyming text and colorful illustrations. Di Donato Roos has also written two other books for children, *How Do Mermaids Poo?* and *The Magic Box*, both of which are also worth checking out.

### **Mine!: How the Hidden Rules of Ownership Control Our Lives, by Michael Heller and James Salzman**

*Recommendation written by [Bryan Behrenshausen](#)*

"A lot of what you know about ownership is wrong," authors Michael Heller and James Salzman write in *Mine!* It's the kind of confrontational invitation people drawn to open source can't help but accept. And this book is certainly one for open source aficionados, whose views on ownership—of code, of ideas, of intellectual property of all kinds—tend to differ from mainstream opinions and received wisdom. In this book, Heller and Salzman lay out the "hidden rules of ownership" that govern who controls access to what. These rules are subtle, powerful, deeply historical conventions that have become so commonplace they just seem incontrovertible. We know this because they've become platitudes: "First come, first served" or "You reap what you sow." Yet we see them play out everywhere: On airplanes in fights over precious legroom, in the streets as neighbors scuffle over freshly shoveled parking spaces, and in courts as juries decide who controls your inheritance and your DNA. Could alternate theories of ownership create space for rethinking some essential rights in the digital age? The authors certainly think so. And if they're correct, we might respond: Can open source software serve as a model for how ownership works—or doesn't—in the future?

**Not All Fairy Tales Have Happy Endings: The Rise and Fall of Sierra On-Line, by Ken Williams**

*Recommendation written by [Joshua Allen Holm](#)*

During the 1980s and 1990s, Sierra On-Line was a juggernaut in the computer software industry. From humble beginnings, this company, founded by Ken and Roberta Williams, published many iconic computer games. King's Quest, Space Quest, Quest for Glory, Leisure Suit Larry, and Gabriel Knight are just a few of the company's biggest franchises.

*Not All Fairy Tales Have Happy Endings* covers everything from the creation of Sierra's first game, [Mystery House](#), to the company's unfortunate and disastrous acquisition by CUC International and the aftermath. The Sierra brand would live on for a while after the acquisition, but the Sierra founded by the Williams was no more. Ken Williams recounts the entire history of Sierra in a way that only he could. His chronological narrative is interspersed with chapters providing advice about management and computer programming. Ken Williams had been out of the industry for many years by the time he wrote this book, but his advice is still extremely relevant.

Sierra On-Line is no more, but the company made a lasting impact on the computer gaming industry. *Not All Fairy Tales Have Happy Endings* is a worthwhile read for anyone interested in the history of computer software. Sierra On-Line was at the forefront of game development

during its heyday, and there are many valuable lessons to learn from the man who led the company during those exciting times.

### **The Soul of a New Machine, by Tracy Kidder**

*Recommendation written by [Gaurav Kamathe](#)*

I am an avid reader of the history of computing. It's fascinating to know how these intelligent machines that we have become so dependent on (and often take for granted) came into being. I first heard of [The Soul of a New Machine](#) via [Bryan Cantrill's blog post](#). This is a non-fiction book written by [Tracy Kidder](#) and published in 1981 for which he [won a Pulitzer prize](#). Imagine it's the 1970s, and you are part of the engineering team tasked with designing the [next generation computer](#). The backdrop of the story begins at Data General Corporation, a then mini-computer vendor who was racing against time to compete with the 32-bit VAX computers from Digital Equipment Corporation (DEC). The book outlines how two competing teams within Data General, both wanting to take a shot at designing the new machine, results in a feud. What follows is a fascinating look at the events that unfold. The book provides insights into the minds of the engineers involved, the management, their work environment, the technical challenges they faced along the way and how they overcame them, how stress affected their personal lives, and much more. Anybody who wants to know what goes into making a computer should read this book.

---

Joshua Allen Holm is an advocate for open access, open educational resources, and open source software. He holds a master's degree in library and information science from Wayne State University and a master's degree in higher education from Grand Valley State University.



# A programmer's guide to GNU C Compiler

By Jayashree Huttanagoudar

C is a well-known programming language, popular with experienced and new programmers alike. Source code written in C uses standard English terms, so it's considered human-readable. However, computers only understand binary code. To convert code into machine language, you use a tool called a *compiler*.

A very common compiler is GCC (GNU C Compiler). The compilation process involves several intermediate steps and adjacent tools.

## Install GCC

To confirm whether GCC is already installed on your system, use the `gcc` command:

```
$ gcc --version
```

If necessary, install GCC using your packaging manager. On Fedora-based systems, use `dnf`:

```
$ sudo dnf install gcc libgcc
```

On Debian-based systems, use `apt`:

```
$ sudo apt install build-essential
```

After installation, if you want to check where GCC is installed, then use:

```
$ whereis gcc
```

## Simple C program using GCC

Here's a simple C program to demonstrate how to compile code using GCC. Open your favorite text editor and paste in this code:

```
// hellogcc.c
#include <stdio.h>
int main() {
    printf("Hello, GCC!\n");
    return 0;
}
```

Save the file as `hellogcc.c` and then compile it:

```
$ ls
hellogcc.c
$ gcc hellogcc.c
$ ls -l
a.out
hellogcc.c
```

As you can see, `a.out` is the default executable generated as a result of compilation. To see the output of your newly-compiled application, just run it as you would any local binary:

```
$ ./a.out
Hello, GCC!
```

## Name the output file

The filename `a.out` isn't very descriptive, so if you want to give a specific name to your executable file, you can use the `-o` option:

```
$ gcc -o hellogcc hellogcc.c
$ ls
a.out  hellogcc  hellogcc.c
$ ./hellogcc
Hello, GCC!
```

This option is useful when developing a large application that needs to compile multiple C source files.

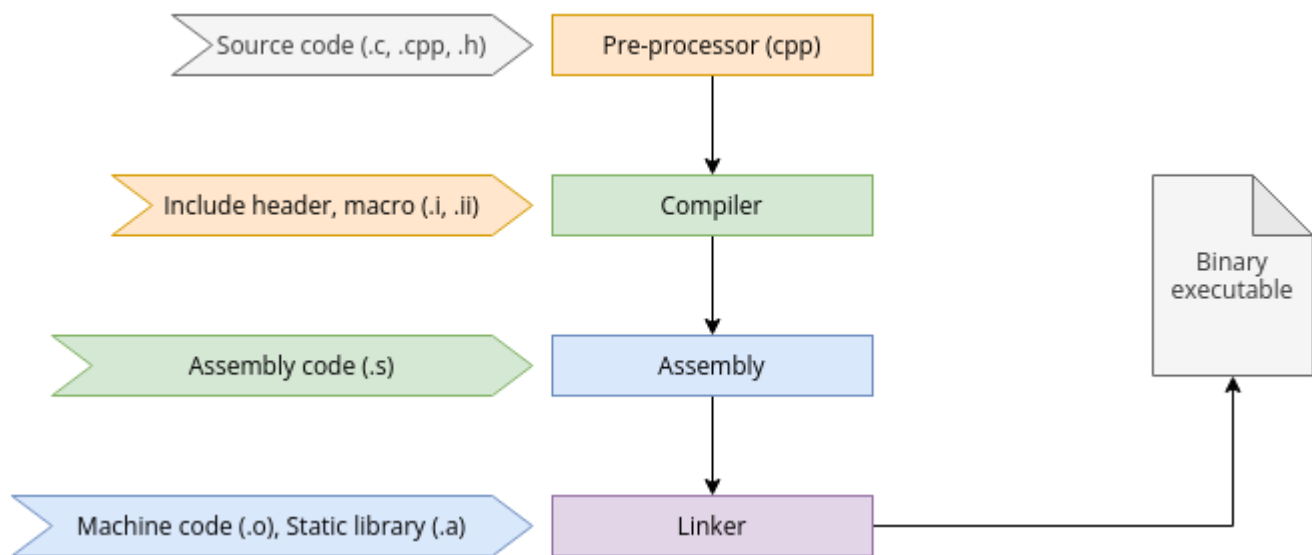
## Intermediate steps in GCC compilation

There are actually four steps to compiling, even though GCC performs them automatically in simple use-cases.

1. Pre-Processing: The GNU C Preprocessor (`cpp`) parses the headers (**#include** statements), expands macros (**#define** statements), and generates an intermediate file such as `hellogcc.i` with expanded source code.
2. Compilation: During this stage, the compiler converts pre-processed source code into assembly code for a specific CPU architecture. The resulting assembly file is named with a `.s` extension, such as `hellogcc.s` in this example.
3. Assembly: The assembler (`as`) converts the assembly code into machine code in an object file, such as `hellogcc.o`.
4. Linking: The linker (`ld`) links the object code with the library code to produce an executable file, such as `hellogcc`.

When running GCC, use the `-v` option to see each step in detail.

```
$ gcc -v -o hellogcc hellogcc.c
```



(Jayashree Huttanagoudar, CC BY-SA 4.0)

## Manually compile code

It can be useful to experience each step of compilation because, under some circumstances, you don't need GCC to go through all the steps.

First, delete the files generated by GCC in the current folder, except the source file.

```
$ rm a.out hellogcc.o
$ ls
hellogcc.c
```

## Pre-processor

First, start the pre-processor, redirecting its output to `hellogcc.i`:

```
$ cpp hellogcc.c > hellogcc.i
$ ls
hellogcc.c  hellogcc.i
```

Take a look at the output file and notice how the pre-processor has included the headers and expanded the macros.

## Compiler

Now you can compile the code into assembly. Use the `-S` option to set GCC just to produce assembly code.

```
$ gcc -S hellogcc.i
$ ls
hellogcc.c  hellogcc.i  hellogcc.s
$ cat hellogcc.s
```

Take a look at the assembly code to see what's been generated.

## Assembly

Use the assembly code you've just generated to create an object file:

```
$ as -o hellogcc.o hellogcc.s
$ ls
hellogcc.c  hellogcc.i  hellogcc.o  hellogcc.s
```

## Linking

To produce an executable file, you must link the object file to the libraries it depends on. This isn't quite as easy as the previous steps, but it's educational:

```
$ ld -o hellogcc hellogcc.o
ld: warning: cannot find entry symbol _start; defaulting to 0000000000401000
```

```
ld: hellogcc.o: in function `main':
hellogcc.c:(.text+0xa): undefined reference to `puts'
```

An error referencing an `undefined puts` occurs after the linker is done looking at the `libc.so` library. You must find suitable linker options to link the required libraries to resolve this. This is no small feat, and it's dependent on how your system is laid out.

When linking, you must link code to core runtime (CRT) objects, a set of subroutines that help binary executables launch. The linker also needs to know where to find important system libraries, including `libc` and `libgcc`, notably within special start and end instructions. These instructions can be delimited by the `--start-group` and `--end-group` options or using paths to `crtbegin.o` and `crtend.o`.

This example uses paths as they appear on a RHEL 8 install, so you may need to adapt the paths depending on your system.

```
$ ld -dynamic-linker \
/lib64/ld-linux-x86-64.so.2 \
-o hello \
/usr/lib64/crt1.o /usr/lib64/crti.o \
--start-group \
-L/usr/lib/gcc/x86_64-redhat-linux/8 \
-L/usr/lib64 -L/lib64 hello.o \
-lgcc \
--as-needed -lgcc_s \
--no-as-needed -lc -lgcc \
--end-group
/usr/lib64/crtn.o
```

The same linker procedure on Slackware uses a different set of paths, but you can see the similarity in the process:

```
$ ld -static -o hello \
-L/usr/lib64/gcc/x86_64-slackware-linux/11.2.0/ \
/usr/lib64/crt1.o /usr/lib64/crti.o \
hello.o /usr/lib64/crtn.o \
--start-group -lc -lgcc -lgcc_eh \
--end-group
```

Now run the resulting executable:

```
$ ./hello
Hello, GCC!
```

## Some helpful utilities

Below are a few utilities that help examine the file type, symbol table, and the libraries linked with the executable.

Use the `file` utility to determine the type of file:

```
$ file hellogcc.c
hellogcc.c: C source, ASCII text
$ file hellogcc.o
hellogcc.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
$ file hellogcc
hellogcc: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=bb76b241d7d00871806e9fa5e814fee276d5bd1a, for GNU/Linux 3.2.0, not
stripped
```

Use the `nm` utility to list symbol tables for object files:

```
$ nm hellogcc.o
0000000000000000 T main
                 U puts
```

Use the `ldd` utility to list dynamic link libraries:

```
$ ldd hellogcc
linux-vdso.so.1 (0x00007ffe3bdd7000)
libc.so.6 => /lib64/libc.so.6 (0x00007f223395e000)
/lib64/ld-linux-x86-64.so.2 (0x00007f2233b7e000)
```

## Wrap up

In this article, you learned the various intermediate steps in GCC compilation and the utilities to examine the file type, symbol table, and libraries linked with an executable. The next time you use GCC, you'll understand the steps it takes to produce a binary file for you, and when something goes wrong, you know how to step through the process to resolve problems.



Jayashree Huttanagoudar is a Senior Software Engineer at Red Hat India Pvt Ltd. She works with Middleware OpenJDK team. She is always curious to learn new things, which adds to her work.

# Analyze the Linux kernel with `ftrace`

By Gaurav Kamathe

An operating system's kernel is one of the most elusive pieces of software out there. It's always there running in the background from the time your system gets turned on. Every user achieves their computing work with the help of the kernel, yet they never interact with it directly. The interaction with the kernel occurs by making system calls or having those calls made on behalf of the user by various libraries or applications that they use daily.

I've covered how to trace system calls in an earlier article using [strace](#). However, with `strace`, your visibility is limited. It allows you to view the system calls invoked with specific parameters and, after the work gets done, see the return value or status indicating whether they passed or failed. But you had no idea what happened inside the kernel during this time. Besides just serving system calls, there's a lot of other activity happening inside the kernel that you're oblivious to.

## Ftrace Introduction

This article aims to shed some light on tracing the kernel functions by using a mechanism called `ftrace`. It makes kernel tracing easily accessible to any Linux user, and with its help you can learn a lot about Linux kernel internals.

The default output generated by the `ftrace` is often massive, given that the kernel is always busy. To save space, I've kept the output to a minimum and, in many cases truncated the output entirely.

I am using Fedora for these examples, but they should work on any of the latest Linux distributions.

## Enabling ftrace

Ftrace is part of the Linux kernel now, and you no longer need to install anything to use it. It is likely that, if you are using a recent Linux OS, ftrace is already enabled. To verify that the ftrace facility is available, run the mount command and search for tracefs. If you see output similar to what is below, ftrace is enabled, and you can easily follow the examples in this article. These commands must be run as the root user (sudo is insufficient.)

```
# mount | grep tracefs
none on /sys/kernel/tracing type tracefs (rw,relatime,seclabel)
```

To make use of ftrace, you first must navigate to the special directory as specified in the mount command above, from where you'll run the rest of the commands in the article:

```
# cd /sys/kernel/tracing
```

## General work flow

First of all, you must understand the general workflow of capturing a trace and obtaining the output. If you're using ftrace directly, there isn't any special ftrace-specific commands to run. Instead, you basically write to some files and read from some files using standard command-line Linux utilities.

The general steps:

1. Write to some specific files to enable/disable tracing.
2. Write to some specific files to set/unset filters to fine-tune tracing.
3. Read generated trace output from files based on 1 and 2.
4. Clear earlier output or buffer from files.
5. Narrow down to your specific use case (kernel functions to trace) and repeat steps 1, 2, 3, 4.

## Types of available tracers

There are several different kinds of tracers available to you. As mentioned earlier, you need to be in a specific directory before running any of these commands because the files of interest are present there. I use relative paths (as opposed to absolute paths) in my examples.



You can view the contents of the `available_tracers` file to see all the types of tracers available. You can see a few listed below. Don't worry about all of them just yet:

```
# pwd
/sys/kernel/tracing
# cat available_tracers
hwlat blk mmiotrace function_graph wakeup_dl wakeup_rt wakeup function nop
```

Out of all the given tracers, I focus on three specific ones: `function` and `function_graph` to enable tracing, and `nop` to disable tracing.

## Identify current tracer

Usually, by default, the tracer is set to `nop`. That is, "No operation" in the special file `current_tracer`, which usually means tracing is currently off:

```
# pwd
/sys/kernel/tracing
# cat current_tracer
nop
```

## View Tracing output

Before you enable any tracing, take a look at the file where the tracing output gets stored. You can view the contents of the file named `trace` using the [cat](#) command:

```
# cat trace
# tracer: nop
#
# entries-in-buffer/entries-written: 0/0   #P:8
#
#          _-----=> irqsoff
#          / _-----=> need-resched
#          | / _---=> hardirq/softirq
#          || / _--=> preempt-depth
#          ||| /      delay
#          TASK-PID    CPU#  ||||   TIMESTAMP  FUNCTION
#          | |         |   ||||       |         |
```

## Enable function tracer

You can enable your first tracer called `function` by writing `function` to the file `current_tracer` (its earlier content was `nop`, indicating that tracing was off.) Think of this operation as a way of enabling tracing:

```
# pwd
/sys/kernel/tracing
# cat current_tracer
nop
# echo function > current_tracer
# cat current_tracer
function
```

## View updated tracing output for function tracer

Now that you've enabled tracing, it's time to view the output. If you view the contents of the `trace` file, you see a lot of data being written to it continuously. I've piped the output and am currently viewing only the top 20 lines to keep things manageable. If you follow the headers in the output on the left, you can see which task and Process ID are running on which CPU. Toward the right side of the output, you see the exact kernel function running, followed by its parent function. There is also time stamp information in the center:

```
# sudo cat trace | head -20
# tracer: function
#
# entries-in-buffer/entries-written: 409936/4276216   #P:8
#
#          _-----=> irqsoff
#          / _-----=> need_resched
#          | / _---=> hardirq/softirq
#          || / _--=> preempt-depth
#          ||| /      delay
#          TASK-PID   CPU#  ||||   TIMESTAMP  FUNCTION
#          | |       |   ||||   |          |
<idle>-0          [000] d...  2088.841739: tsc_verify_tsc_adjust <-
arch_cpu_idle_enter
<idle>-0          [000] d...  2088.841739: local_touch_nmi <-do_idle
<idle>-0          [000] d...  2088.841740: rcu_nocb_flush_deferred_wakeup
<-do_idle
<idle>-0          [000] d...  2088.841740: tick_check_broadcast_expired <-
do_idle
<idle>-0          [000] d...  2088.841740: cpuidle_get_cpu_driver <-
do_idle
```

```

<idle>-0      [000] d... 2088.841740: cpuidle_not_available <-do_idle
<idle>-0      [000] d... 2088.841741: cpuidle_select <-do_idle
<idle>-0      [000] d... 2088.841741: menu_select <-do_idle
<idle>-0      [000] d... 2088.841741: cpuidle_governor_latency_req <-
menu_select

```

Remember that tracing is on, which means the output of tracing continues to get written to the trace file until you turn tracing off.

## Turn off tracing

Turning off tracing is simple. All you have to do is replace `function` tracer with `nop` in the `current_tracer` file and tracing gets turned off:

```

# cat current_tracer
function
# echo nop > current_tracer
# cat current_tracer
nop

```

## Enable function\_graph tracer

Now try the second tracer, called `function_graph`. You can enable this using the same steps as before: write `function_graph` to the `current_tracer` file:

```

# echo function_graph > current_tracer
# cat current_tracer
function_graph

```

## Tracing output of function\_graph tracer

Notice that the output format of the trace file has changed. Now, you can see the CPU ID and the duration of the kernel function execution. Next, you see curly braces indicating the beginning of a function and what other functions were called from inside it:

```

# cat trace | head -20
# tracer: function_graph
#
# CPU    DURATION                FUNCTION CALLS
# |      |      |                | | | |
6)      |      |                n_tty_write() {
6)      |      |                down_read() {

```

```

6)      |      __cond_resched() {
6) 0.341 us |      rcu_all_qs();
6) 1.057 us |      }
6) 1.807 us |      }
6) 0.402 us |      process_echoes();
6)      |      add_wait_queue() {
6) 0.391 us |      _raw_spin_lock_irqsave();
6) 0.359 us |      _raw_spin_unlock_irqrestore();
6) 1.757 us |      }
6) 0.350 us |      tty_hung_up_p();
6)      |      mutex_lock() {
6)      |      __cond_resched() {
6) 0.404 us |      rcu_all_qs();
6) 1.067 us |      }

```

## Enable trace settings to increase the depth of tracing

You can always tweak the tracer slightly to see more depth of the function calls using the steps below. After which, you can view the contents of the trace file and see that the output is slightly more detailed. For readability, the output of this example is omitted:

```

# cat max_graph_depth
0
# echo 1 > max_graph_depth ## or:
# echo 2 > max_graph_depth
# sudo cat trace

```

## Finding functions to trace

The steps above are sufficient to get started with tracing. However, the amount of output generated is enormous, and you can often get lost while trying to find out items of interest. Often you want the ability to trace specific functions only and ignore the rest. But how do you know which processes to trace if you don't know their exact names? There is a file that can help you with this—`available_filter_functions` provides you with a list of available functions for tracing:

```

# wc -l available_filter_functions
63165 available_filter_functions

```

## Search for general kernel functions

Now try searching for a simple kernel function that you are aware of. User-space has `malloc` to allocate memory, while the kernel has its `kmalloc` function, which provides similar functionality. Below are all the `kmalloc` related functions:

```
# grep kmalloc available_filter_functions
debug_kmalloc
mempool_kmalloc
kmalloc_slab
kmalloc_order
kmalloc_order_trace
kmalloc_fix_flags
kmalloc_large_node
__kmalloc
__kmalloc_track_caller
__kmalloc_node
__kmalloc_node_track_caller
[...]
```

## Search for kernel module or driver related functions

From the output of `available_filter_functions`, you can see some lines ending with text in brackets, such as `[kvm_intel]` in the example below. These functions are related to the kernel module `kvm_intel`, which is currently loaded. You can run the `lsmod` command to verify:

```
# grep kvm available_filter_functions | tail
__pi_post_block [kvm_intel]
vmx_vcpu_pi_load [kvm_intel]
vmx_vcpu_pi_put [kvm_intel]
pi_pre_block [kvm_intel]
pi_post_block [kvm_intel]
pi_wakeup_handler [kvm_intel]
pi_has_pending_interrupt [kvm_intel]
pi_update_irte [kvm_intel]
vmx_dump_dtsel [kvm_intel]
vmx_dump_sel [kvm_intel]
# lsmod | grep -i kvm
kvm_intel          335872  0
kvm                987136  1 kvm_intel
irqbypass         16384   1 kvm
```

## Trace specific functions only

To enable tracing of specific functions or patterns, you can make use of the `set_ftrace_filter` file to specify which functions from the above output you want to trace.

This file also accepts the `*` pattern, which expands to include additional functions with the given pattern. As an example, I am using the `ext4` filesystem on my machine. I can specify `ext4` specific kernel functions to trace using the following commands:

```
# mount | grep home
/dev/mapper/fedora-home on /home type ext4 (rw,relatime,seclabel)
# pwd
/sys/kernel/tracing
# cat set_ftrace_filter
#### all functions enabled ####
$
$ echo ext4_* > set_ftrace_filter
$
$ cat set_ftrace_filter
ext4_has_free_clusters
ext4_validate_block_bitmap
ext4_get_group_number
ext4_get_group_no_and_offset
ext4_get_group_desc
[...]
```

Now, when you see the tracing output, you can only see functions `ext4` related to kernel functions for which you had set a filter earlier. All the other output gets ignored:

```
# cat trace |head -20
## tracer: function
#
# entries-in-buffer/entries-written: 3871/3871   #P:8
#
#          _-----=> irqs-off
#          / _-----=> need-resched
#          | / _---=> hardirq/softirq
#          || / _--=> preempt-depth
#          ||| /      delay
# TASK-PID   CPU#  ||||   TIMESTAMP  FUNCTION
#   | |       |   ||||   |         |
cupsd-1066   [004] ....   3308.989545: ext4_file_getattr <-vfs_fstat
cupsd-1066   [004] ....   3308.989547: ext4_getattr <-ext4_file_getattr
cupsd-1066   [004] ....   3308.989552: ext4_file_getattr <-vfs_fstat
```

```
cupsd-1066 [004] .... 3308.989553: ext4_getattr <-ext4_file_getattr
cupsd-1066 [004] .... 3308.990097: ext4_file_open <-do_dentry_open
cupsd-1066 [004] .... 3308.990111: ext4_file_getattr <-vfs_fstat
cupsd-1066 [004] .... 3308.990111: ext4_getattr <-ext4_file_getattr
cupsd-1066 [004] .... 3308.990122: ext4_llseek <-ksys_llseek
cupsd-1066 [004] .... 3308.990130: ext4_file_read_iter <-new_sync_read
```

## Exclude functions from being traced

You don't always know what you want to trace but, you surely know what you don't want to trace. For that, there is this file aptly named `set_ftrace_notrace`—notice the "no" in there. You can write your desired pattern in this file and enable tracing, upon which everything except the mentioned pattern gets traced. This is often helpful to remove common functionality that clutters our output:

```
# cat set_ftrace_notrace
##### no functions disabled #####
```

## Targetted tracing

So far, you've been tracing everything that has happened in the kernel. But that won't help us if you wish to trace events related to a specific command. To achieve this, you can turn tracing on and off on-demand and, and in between them, run our command of choice so that you do not get extra output in your trace output. You can enable tracing by writing `1` to `tracing_on`, and `0` to turn it off:

```
# cat tracing_on
0
# echo 1 > tracing_on
# cat tracing_on
1
### Run some specific command that we wish to trace here ###
# echo 0 > tracing_on
# cat tracing_on
0
```

## Tracing specific PID

If you want to trace activity related to a specific process that is already running, you can write that PID to a file named `set_ftrace_pid` and then enable tracing. That way, tracing is limited to this PID only, which is very helpful in some instances:

```
# echo $PID > set_ftrace_pid
```

## Conclusion

Ftrace is a great way to learn more about the internal workings of the Linux kernel. With some practice, you can learn to fine-tune `ftrace` and narrow down your searches. To understand `ftrace` in more detail and its advanced usage, see these excellent articles written by the core author of `ftrace` himself—Steven Rostedt.



*Seasoned Software Engineering professional.  
Primary interests are Security, Linux, Malware.  
Loves working on the command-line.  
Interested in low-level software and  
understanding how things work.  
Opinions expressed here are my own and not  
that of my employer*



# 5 ways to involve people who don't write code in the DevOps process

By Will Kelly

DevOps transformation extends beyond development and operations teams. It's also relevant to other parts of the organization. These new collaborators can offer new insights to the development team seeking to maintain alignment with customer needs.

This article describes ways to involve other parts of your business in DevOps.

## Collaborate with marketing and sales teams

Product and services companies need to account for their sales and marketing teams in their DevOps transformation. DevOps can help bring down the silos between sales, marketing, and development.

Sales and marketing teams supporting new product launches need constant visibility into development project progress. I'd also argue developer stakeholders earn a view of marketing activities. The days of technically inaccurate surprises in marketing collateral should be no more in a DevOps culture. Sales organizations can communicate customer feedback and requirements into the development cycle so that incremental releases can include customer-requested features. You can increase this involvement in a couple of ways:

- Use [open source group chat tools](#) such as [Mattermost](#) or [Rocket.chat](#) to set up go-to-market (GTM) or product release channels for the sharing of product information if you haven't already.
- Include sales and marketing in the build release cycle, allowing them to receive a demo or even use a test version.
- Give your sales and marketing teams access to documentation during the development cycle.

Automation is a priority in DevOps. It's up to you to educate your marketing team on how automation changes how your organization delivers software internally and externally, so it becomes part of your corporate story. I also suggest examining ways to automate data reporting from your continuous integration/continuous development (CI/CD) toolchain to benefit marketing.

Collaboration and shared roles with marketing in user story development, product definitions, user documentation, and marketing collateral create new meaning with a project schedule to support such activities.

## Accelerate and automate technical content creation

The time has come for technical writers and other content developers to sit at the DevOps table. It's a subject I covered in a previous [Opensource.com article](#).

Here are some ways to involve technical and marketing content creators in your DevOps process:

- Knockdown any cultural or scheduling silos between your technical writer and the project team and give the writer a seat at the table, starting by embedding the writer on the team.
- Manage any project-related content with a similar toolchain or pipeline model as you manage your continuous delivery using automated documentation tools such as [docToolchain](#), [Hugo](#), or [Jekyll](#).
- Take a [progressionist](#) versus perfectionist approach to content, so content development keeps pace with product development.
- Introduce a [continuous documentation model](#) (or a similar methodology) to accelerate content development to DevOps velocity.

## Seek licensing guidance from your legal department

Even your legal and finance department has a potential role in your DevOps process. While a corporate attorney may not be billing their time directly to your DevOps projects, there's work for them in some facets of your open source and commercial software licensing. Here are some ways to involve your legal department in DevOps:

- Make open source software licensing a responsibility of one of your corporate attorneys and encourage collaboration between legal and your open source savvy developers.
- Mid-sized enterprises should include legal representation as part of any open source center of excellence or other cross-functional teams that focus on open source software.
- Large-sized enterprises should include legal representation as part of their [open source program office](#).

## Monitor costs with your accounting and finance department

[Financial Operations \(FinOps\)](#) and cloud cost optimization are integral elements of digital transformation projects these days. Cost [monitoring](#) and optimization should be part of any Kubernetes deployment.

Setting aside the tools selection to make Kubernetes cost monitoring a reality, you need to start collaborating with your accounting and finance department early. An engineer or solution architect should partner with an accounting department representative to put financial controls and monitoring into the planning phase of DevOps.

## Implement self-serve reporting for your executives

Every so often, you may have to involve your executive team in your DevOps process. No, you don't want them working shoulder-to-shoulder with your teams. Instead, you want to put your backend data to work through the judicious and innovative use of analytics and reporting.

Some examples include:

- Application change time, the time between code commit and deployment.
- Issue volume, capturing the number of issues that staff and customers report in a given period.
- Time to value is a business measurement of the time spent between a feature request and business value realization.

When you put in the proper reporting and audit trails for your DevOps efforts, you also gain a new channel to communicate project successes and challenges to stakeholders. More importantly, you now have actionable data to back up what you report.

## Final thoughts

Ultimately, DevOps can and should transform your business. Full-scale transformation isn't happening unless your development and operations teams begin collaborating with other business units to achieve corporate-wide DevOps goals. While many organizations speak to the power of cross-functional teams, involving people who don't write code in the DevOps process is ultimately a cultural exercise because it brings down even more silos across your organization.

---



Will Kelly is a product marketer and writer. His career has been spent writing bylined articles, white papers, marketing collateral, and technical content about the cloud and DevOps.

Opensource.com, TechTarget, InfoQ, and others have published his articles about DevOps and the cloud. He lives and works in the Northern Virginia area.

Follow him on Twitter: [@willkelly](https://twitter.com/willkelly)

# 9 open source alternatives to try in 2022

By Lauren Maffeo

2021 was another year spent largely online, but that's nothing new for the open source world. The ability to work from anywhere is in our DNA, preceding the pandemic that ushered remote work into the mainstream.

Still, all that time in front of screens this year made our community consider open source alternatives. Regardless of the tool type you need, many of the most popular vendors are not your only option.

If you're burned out on Zoom, want a CRM that's not Salesforce, or would like an analytics tool that Google doesn't own, read on. We've got the most popular articles on open source alternatives that readers loved in 2021.

## My favorite open source project management tools

If you're managing a project, it might seem like you have an endless amount of tools to choose from. With Gantt charts crossing into Agile territory, it's now more common to see them deployed in service of large projects.

If you think Microsoft Project is your only option, fear not. Frank Bergmann shares several [open source alternatives](#) for single users to plan and track large, single projects. Redmine, ProjectLibre, and TaskJuggler are among the open source alternatives covered in this list.

## An open source alternative to Microsoft Exchange

Microsoft Exchange's dominance in groupware might be coming to an end. In 2020, an Austrian open source developer built grommunio to serve as an open source alternative to Exchange.

In his [review of grommunio](#), Markus Feilner shares the vast list of features that the tool offers. Integrated, native exchange protocols let Outlook and smartphones connect to grommunio the same way they'd connect to Exchange. Calendar management, video conferencing, and meeting capabilities courtesy of [Jitsi](#) are among this robust list.

## Manage your budget on Linux with this open source finance tool

When it comes to money management, Linux likely isn't the first tool platform you think of. However, it turns out that there are many apps built on Linux to help keep your personal finances on track.

Options like HomeBank and KMyMoney let you import data from your bank and review expenses against your budget. Seth Kenlon, [this article's author](#), prefers Skrooge and shares how he uses it. If you're looking for an open source tool to track expenditures, this could be the one.

## 5 open source alternatives to Zoom

Almost two years into the pandemic, it's safe to say most of us are all Zoom-ed out. For many of us, most work and social events are exclusively online. But as Seth Kenlon astutely points out, one of open source's biggest benefits is the ability to work remotely. Accordingly, [open source enthusiasts have several choices](#) if you're sick of Zoom.

Popular names like Jitsi make this list, along with some surprises. For instance, did you know that Signal added group video calls to its features list? P2p.chart, BigBlueButton, and Wire also make this list of tools based on various video needs, from small group calls to corporate meetings.

## Schedule appointments with an open source alternative to Doodle

Appointment scheduling apps like Doodle save the song and dance of picking mutually available dates. For folks like Kevin Sonney, who hosts a podcast, these tools help him and each guest easily find dates that suit them both.

While Doodle wins the popular vote, Sonney shares his experience with Easy!Appointments [in this article](#). Aimed at helping service organizations, Easy!Appointments has a WordPress plug-

in that lets users put request forms on pages or posts. The app also syncs with Google Calendar, and there's talk of adding support to sync with additional backends.

## **Why choose Plausible for an open source alternative to Google Analytics**

If you need to use web analytics, it might seem like Google Analytics is your only option. Uku Taht and Marko Saric set out to change that when they built Plausible.io to provide an open source analytics tool that could manage large amounts of data without a performance decline. Two years post-release, Plausible can ingest over 80 million records per month.

In this article, Ben Rometsch shares [Plausible's journey](#) from software with sensitive code to an open source option under AGPL. It's well worth the read if you're curious how this little analytics engine that could grew from 500 to 4300 stars on GitHub.

## **Try Chatwoot, an open source customer relationship platform**

Looking for an open source end-to-end platform that covers ticket management/support? Built with Ruby and Vue.js, Chatwoot might be a viable choice if you want something other than Salesforce or Zendesk.

Nitish Tiwari shares Chatwoot's architecture, installation, and key features in [this piece](#). It's available on several platforms, including Linux and Docker. This article shares the installation process for Docker, along with features like channels and integrations.

## **Get started with an open source customer data platform**

If you're managing data at scale, you likely use or are searching for a data warehouse. Those searching for an open source warehouse can consider RudderStack, which collects and routes event stream data before building customer data lakes on data warehouses.

Amey Varangaonkar [shares how to get RudderStack's workspace token](#), then install and deploy on Docker or Kubernetes. The tool's rudder-server repository and destination integrations are open source, letting users see how the utility completes complicated tasks. In an era of risky black boxes, having this transparency is a big benefit.

## Try Dolibarr, an open source customer relationship management platform

You're not seeing things: This is another article on open source alternatives to customer relationship management. [This piece](#) by Pradeep Vijayakumar covers Dolibarr, which has robust ERP features along with its CRM status.

Vijayakumar works directly on Dolibarr, so his knowledge of installing its CRM, adding customer data, and setting up campaigns is unparalleled. Screenshots show each process in detail, with helpful hints to optimize the tool.



Lauren Maffeo has reported on and worked within the global technology sector. She started her career as a freelance journalist covering tech trends for The Guardian and The Next Web from London. Today, she works as a service designer for Steampunk, a human-centered design firm building civic tech solutions for government agencies. Prior to Steampunk, Lauren was an associate principal analyst at Gartner, where

she covered the impact of emerging tech like AI and blockchain on small and midsize business owners.

An advocate of open access publishing, Lauren serves as a founding editor of Springer's AI and Ethics journal, the first multidisciplinary journal on this subject. She is also an area editor for the open access Data and Policy journal with Cambridge University Press. Lauren has spoken at open source events including the Open Source Summits in North America and Europe, All Things Open, DrupalCon North America, DrupalGovCon, and GitHub Universe.



# Share your Linux terminal with tmate

By Sumantra Mukherjee

As a member of the Fedora Linux QA team, I sometimes find myself executing a bunch of commands that I want to broadcast to other developers. If you've ever used a [terminal multiplexer](#) like [tmux](#) or [GNU Screen](#), you might think that that's a relatively easy task. But not all of the people I want to see my demonstration are connecting to my terminal session from a laptop or desktop. Some might have casually opened it from their phone browser—which they can readily do because I use [tmate](#).

## Linux terminal sharing with tmate

Watching someone else work in a Linux terminal is very educational. You can learn new commands, new workflows, or new ways to debug and automate. But it can be difficult to capture what you're seeing so you can try it yourself later. You might resort to taking screenshots or a screen recording of a shared terminal session so you can type out each command later. The only other option is for the person demonstrating the commands to record the session using a tool like [Asciinema](#) or [script and scriptreplay](#).

But with tmate, a user can share a terminal either in read-only mode or over SSH. Both the SSH and the read-only session can be accessed through a terminal or as an HTML webpage.

I use read-only mode when I'm onboarding people for the Fedora QA team because I need to run commands and show the output, but with tmate, folks can keep notes by copying and pasting from their browser to a text editor.

## Linux tmate in action

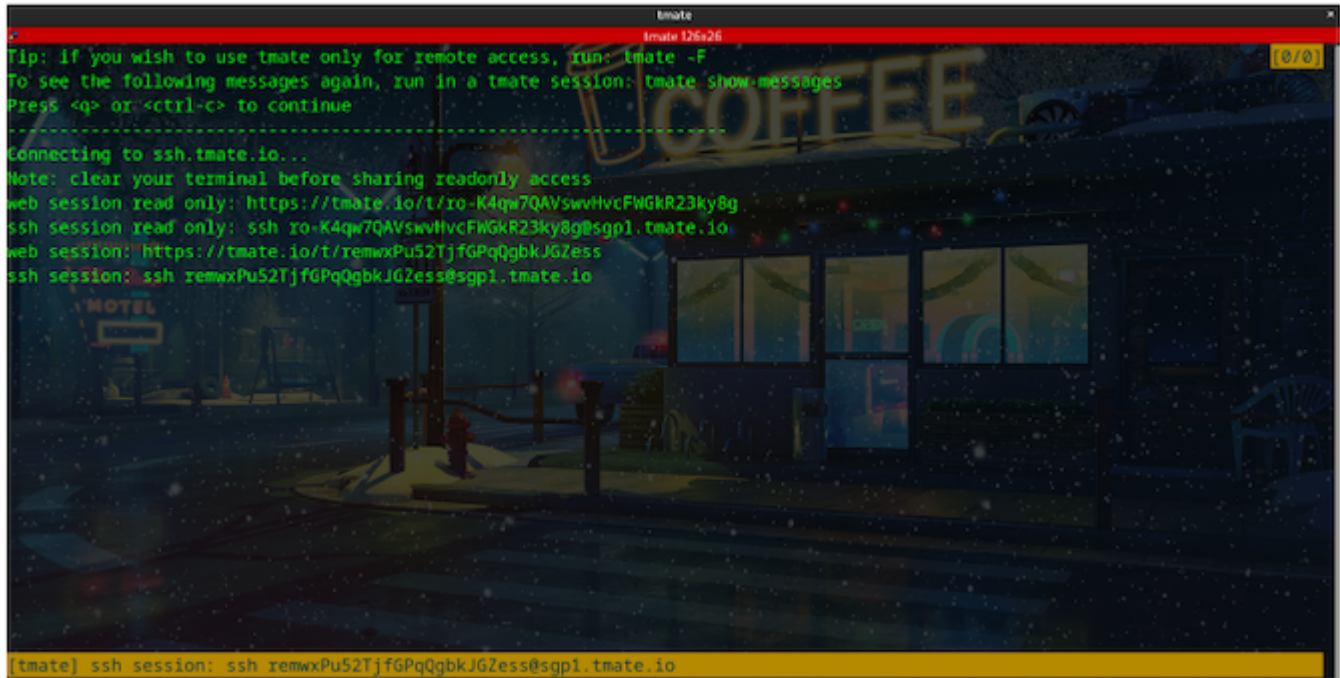
On Linux, you can install tmate with your package manager. For instance, on Fedora:

```
$ sudo dnf install tmate
```

On Debian and similar distributions:

```
$ sudo apt install tmate
```

On macOS, you can install it using [Homebrew](#) or [MacPorts](#). If you need instructions for other Linux distributions, refer to the [install](#) guide.



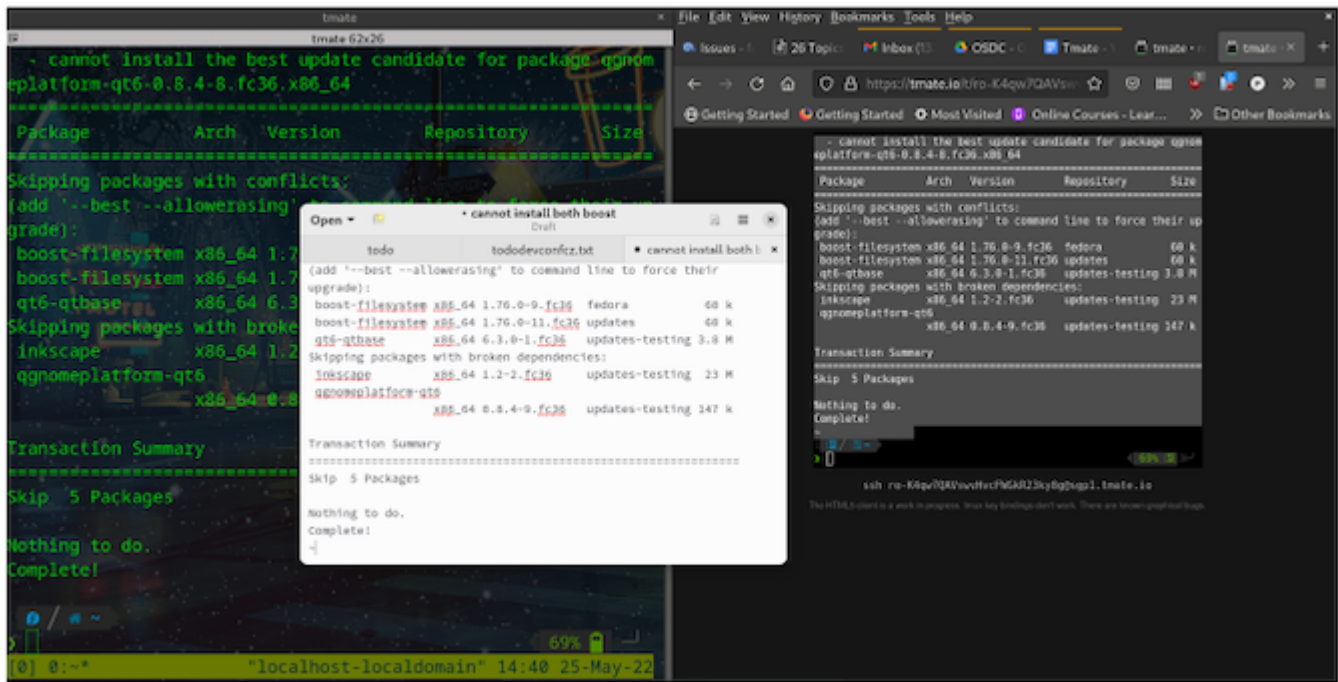
(Sumantro Mukherjee, CC BY-SA 4.0)

Once installed, start tmate:

```
$ tmate
```

When tmate launches, links are generated to provide access to your terminal session over HTTP and SSH. Each protocol features a read-only option as well as a reverse SSH session.

Here's what a web session looks like:



(Sumantro Mukherjee, CC BY-SA 4.0)

Tmate's web console is HTML5, so, as a result, a user can copy the entire screen and paste it into a terminal to run the same commands.

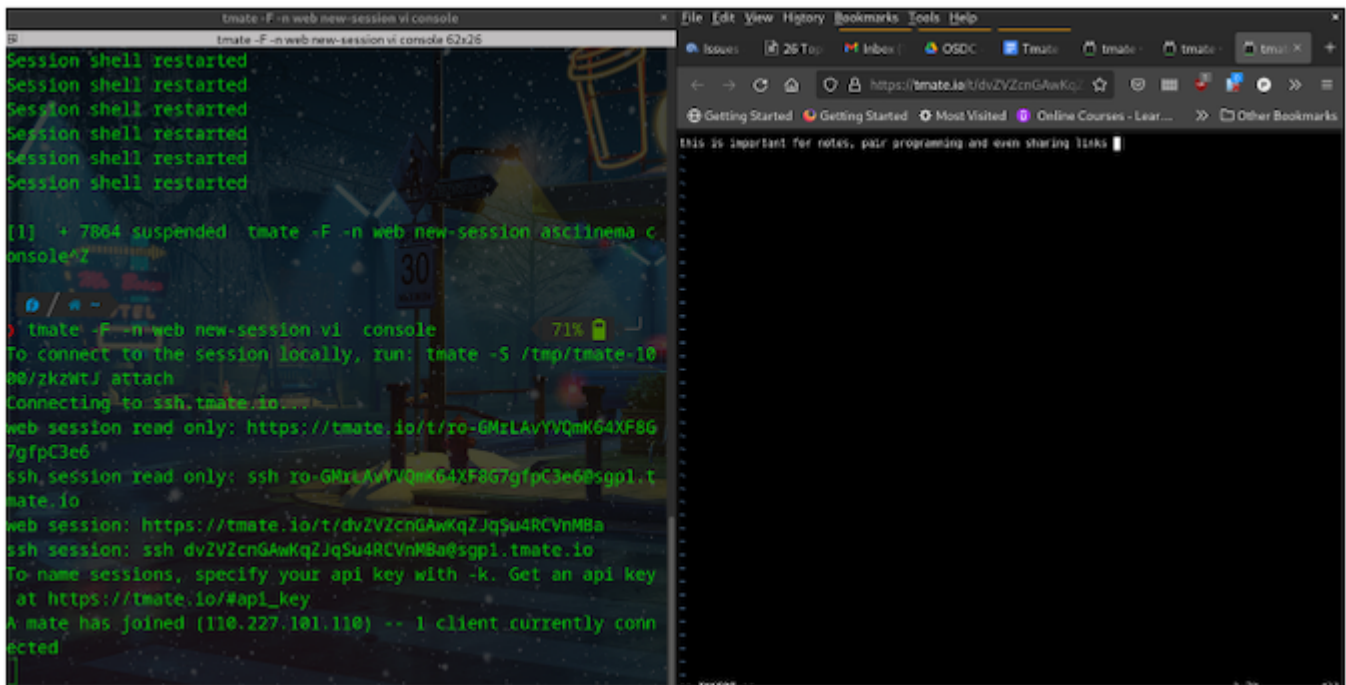
## Keeping a session alive

You may wonder what happens if you accidentally close your terminal. You may also wonder about sharing your terminal with a different console application. After all, tmate is a multiplexer, so it should be able to keep sessions alive, detach and re-attach to a session, and so on.

And of course, that's exactly what tmate can do. If you've ever used tmux, this is probably pretty familiar.

```
$ tmate -F -n web new-session vi console
```

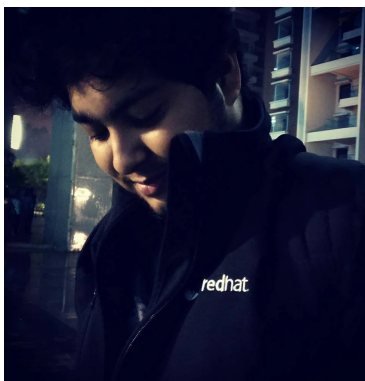
This command opens up `new-session` in `Vi`, and the `-F` option ensures that the session re-spawns even when closed.



(Sumanthro Mukherjee, CC BY-SA 4.0)

## Social multiplexing

Tmate gives you the freedom of tmux or GNU Screen plus the ability to share your sessions with others. It's a valuable tool for teaching other users how to use a terminal, demonstrating the function of a new command, or debugging unexpected behavior. It's open source, so give it a try!



*Hey, open source folks! I am Sumanthro, hailing from India (the eastern part - former capital during the British era AKA Kolkata). I love sharing knowledge and writing about technology and experiences (mostly that I try every day).*

*Anything from k8s, ansible to setting up TCMS for make-believe projects is right up my alley. I work as a Software Developer in Test in Fedora Project and contribute to open source communities in the form of testing, public speaking, documentation, mentorship,*

*and onboarding. Playing Jenga, reading books, and brewing coffee are a few of my hobbies, besides implementing productivity equations in my life and creating vision boards for targets I wanna achieve!*

# What's new with Java 17 and containers?

By Daniel Oh

Container platforms and [edge computing](#) continue to grow, powering major networks and applications across the globe, and Java technologies have evolved new features and improved performance to match steps with modern infrastructure. [Java 17](#) (OpenJDK 17) was released recently (September 2021) with the following major features:

- [Restore Always-Strict Floating-Point Semantics](#)
- [Enhanced Pseudo-Random Number Generators](#)
- [Strongly Encapsulate JDK Internals](#)
- [Pattern Matching for switch \(Preview\)](#)
- [Foreign Function & Memory API \(Incubator\)](#)
- [Vector API \(Second Incubator\)](#)
- [Context-Specific Deserialization Filters](#)

Developers are wondering how to start implementing application logic using the new features of Java 17 and then build and run them on the same OpenJDK 17 runtime. Luckily, [Quarkus](#) enables the developers to scaffold a new application with Java 17. It also provides a [live coding](#) capability that allows developers to focus only on implementing business logic instead of compiling, building, deploying, and restarting runtimes to apply code changes.

**Note:** If you haven't already installed OpenJDK 17, [download a binary](#) on your operating system.

This tutorial teaches you to use the Pseudo-Random Number Generators (PRNG) algorithms of Java 17 on Quarkus. Get started by scaffolding a new project using the [Quarkus command-line tool](#) (CLI):

```
$ quarkus create app prng-example --java=17
```

The output looks like this:

```
...
[SUCCESS] 🍏 quarkus project has been successfully generated in:
--> /Users/danieloh/quarkus-demo/prng-example
...
```

Unlike traditional Java frameworks, Quarkus provides live coding features for developers to rebuild and deploy while the code changes. In the end, this capability accelerates inner loop development for Java developers. Run your Quarkus application using Dev mode:

```
$ cd prng-example
$ quarkus dev
```

The output looks like this:

```
...
INFO [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live Coding
activated.
INFO [io.quarkus] (Quarkus Main Thread) Installed features: [cdi, resteasy,
smallrye-context-propagation, vertx]
--
Tests paused
Press [r] to resume testing, [o] Toggle test output, [:] for the terminal, [h]
for more options>
```

Java 17 enables developers to generate random integers within a specific range based on the **Xoshiro256PlusPlus** PRNG algorithm. Add the following code to the `hello()` method in the `src/main/java/org/acme` directory:

```
RandomGenerator randomGenerator =
RandomGeneratorFactory.of("Xoshiro256PlusPlus").create(999);
for ( int i = 0; i < 10 ; i++) {
    int result = randomGenerator.nextInt(11);
    System.out.println(result);
}
```

Next, invoke the RESTful API (`/hello`) to confirm that random integers are generated. Execute the following cURL command line in your local terminal or access the endpoint URL using a web browser:

```
$ curl localhost:8080/hello
```

Go back to the terminal where you're running Quarkus Dev mode. There you'll see the following ten random numbers:

```
4
6
9
5
7
6
5
0
6
10
```

**Note:** You don't need to rebuild code and restart the Java runtime at all. You'll also see the output Hello RESTEasy in the terminal where you ran the `curl` command line.

## Wrap up

This article shows how Quarkus allows developers to start a new application development based on OpenJDK 17. Furthermore, Quarkus increases developers' productivity by live coding. For a production deployment, developers can make a [native executable](#) based on OpenJDK 17 and [GraalVM](#).



Technical Marketing, Developer Advocate, CNCF Ambassador, Public Speaker, Published Author, Quarkus, Red Hat Runtimes.



# A simple CSS trick for dark mode

By Ayush Sharma

You're likely already familiar with media queries. They're in widespread use for making websites responsive. The `width` and `height` properties contain the viewport's dimensions. You then use CSS to render different layouts at different dimensions.

The [prefers-color-scheme media query](#) works the same way. The user can configure their operating system to use a light or dark theme. `Prefer-color-scheme` contains that value. The value is either `light` or `dark`, though the W3C spec states that it might support future values like `sepia`. I specify different values of CSS variables for both modes and let the user's OS decide.

## The prefers-color-scheme media queries

The two variations of the `prefers-color-scheme` media query are:

```
/* Light mode */
@media (prefers-color-scheme: light) {
  :root {
    --body-bg: #FFFFFF;
    --body-color: #000000;
  }
}
/* Dark mode */
@media (prefers-color-scheme: dark) {
  :root {
    --body-bg: #000000;
    --body-color: #FFFFFF;
  }
}
```



In the above CSS, `--body-bg` and `--body-color` are [CSS variables](#). As you can see, they contain different values for both modes. In the light theme, I'm setting a white background with black text. In the dark theme, I'm setting black background with white text.

Since the [spec](#) says that W3C might introduce future values, it makes sense to convert this CSS into a boolean.

```
/* Light mode */
:root {
  --body-bg: #FFFFFF;
  --body-color: #000000;
}
/* Dark mode */
@media (prefers-color-scheme: dark) {
  :root {
    --body-bg: #000000;
    --body-color: #FFFFFF;
  }
}
```

In the above code, I'm defining a light theme by default and converting it to the dark theme if the media query is `dark`. This way, any future values added to the media query will set the light theme by default.

## Using the CSS variables

Now that I have different values for different themes, I need to actually use them to style the page.

```
body {
  background: var(--body-bg);
  color: var(--body-color);
}
```

The [var\(\) syntax](#) is how CSS uses variables. In the above code, I'm saying set the `background` to the value of `--body-bg` and set the `color` to the value of `--body-color`. Note that the values of these variables are coming from the media query. Meaning that the background and foreground color changes based on the OS's setting!

This is the real power of the media query: Providing a consistent user experience from OS to the web page.

If you go to [findmymastodon.com](https://findmymastodon.com) and switch your OS's theme, you'll see the transition from one theme to the other.

The [CSS Working Group](#) website also uses the same media queries. Change your OS theme, and the website switches themes to adjust.

## Conclusion

Notice that using `prefers-color-scheme` is no different from using a regular programming language. I define variables whose values change based on some logic. And those variables are then used for further operations.

The ability to let your website adjust to the user's theme of choice is a great accessibility feature. And it further blurs the line between desktop and web for the benefit of the user. The latest browser versions [support prefers-color-scheme](#), so you can begin experimenting today.

Happy coding.



I am a writer and AWS Solutions Architect. I work with startups and enterprises on Software Engineering, DevOps, SRE, and Cloud Architecture. I write about my experiences on <https://ayushsharma.in>

# How I automate plant care using Raspberry Pi and open source tools

By Kevin Sonney

*Automation is a hot topic right now. In my day job as an SRE part of my remit is to automate as many repeating tasks as possible. But how many of us do that in our daily, not-work, lives? This year, I am focused on automating away the toil so that we can focus on the things that are important.*

Home Assistant has so many features and integrations, it can be overwhelming at times. And as I've mentioned in previous articles, I use it for many things, including monitoring plants.

```
$ bluetoothctl scan le
Discovery started
[NEW] Device
[NEW] Device
[NEW] Device
[NEW] Device
[NEW] Device
[NEW] Device
[NEW] Device
```

There are numerous little devices you can buy to keep an eye on your plants. The Xiaomi MiaFlora devices are small, inexpensive, and have a native integration with Home Assistant. Which is great—as long as the plant and Home Assistant are in the same room.

We've all been in places where one spot there is a great signal, and moving 1mm in any direction makes it a dead zone—and it is even more frustrating when you are indoors. Most Bluetooth LE (Low Energy) devices have a range of about 100m, but that's using line of sight, and does not include interference from things like walls, doors, windows, or major appliances

(seriously, a refrigerator is a great big signal blocker). Remote Home Assistant is perfect for this. You can set up a Raspberry Pi with Home Assistant Operating System (HASSOS) in the room with the plants, and then use the main Home Assistant as a central control panel. I tried this on a Raspberry Pi Zero W, and while the Pi Zero W can run Home Assistant, it doesn't do it very well. You probably want a Pi 3 or Pi 4 when doing this.

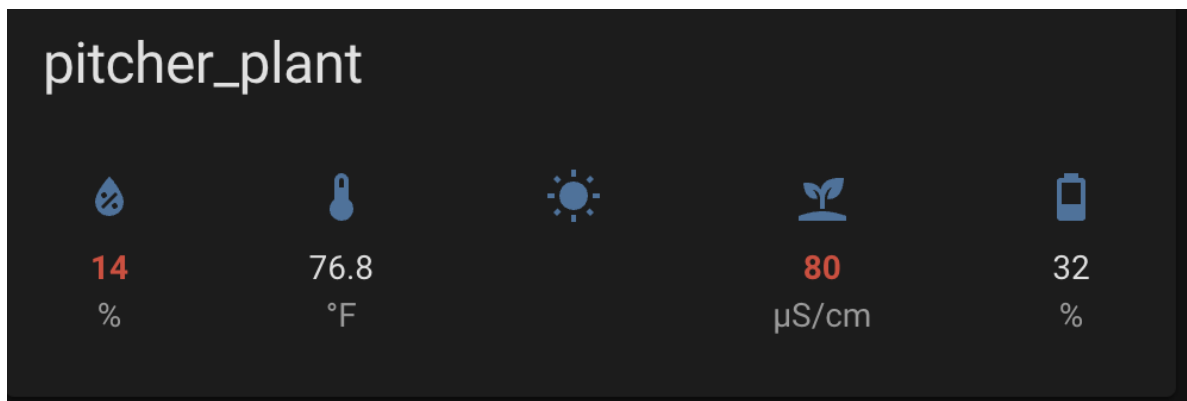
Start with a fresh HASSOS installation, and make sure everything is up-to-date, then install HACS and Remote Home Assistant like I did in my article [Automate and manage multiple devices with Remote Home Assistant](#). Now for the tricky bits. Install the **SSH and Web Terminal** Add-on, and turn off **Protection Mode** so that you can get a session on the base OS and not in a container. Start the add-on, and it appears on the sidebar. Click on it to load the terminal.

You are now in a root session terminal on the Pi. Insert all the warnings here about being careful and how you can mess up the system (you know the ones). Inside the terminal, run `bluetoothctl scan le` to find the plant sensor, often named "Flower Care" like mine.

Make a note of the address for the plant sensor. If you have more than one, it could be confusing to figure out which is which, and can take some trial and error. Once you've identified the plant sensor, it is time to add it to Home Assistant. This requires editing the `configuration.yml` file directly, either with the file editor add on, or in the terminal you just created. In my case, I added both a sensor and a plant block to the configuration.

```
sensor:
- platform: miflora
  scan_interval: 60
  mac: "C4:7C:8D:6C:DE:FE"
  name: "pitcher_plant"
  plant:
    pitcher_plant:
      sensors:
        moisture: sensor.pitcher_plant_moisture
        battery: sensor.pitcher_plant_battery
        temperature: sensor.pitcher_plant_temperature
        conductivity: sensor.pitcher_plant_conductivity
        brightness: sensor.pitcher_plant_brightness
```

Save the file, and restart Home Assistant, and you should see a plant card on the Overview tab.



(Kevin Sonney, CC BY-SA 40)

Once that's done, go back to the main Home Assistant, and add the newly available `plant` component to the list of things to import from the remote. You can then add the component to dashboards on the main HASS installation, and create automations and notifications based on the plant status.

I use this to monitor a pitcher plant, and I have more sensors on the way so I can keep tabs on all my houseplants—all of which live outside the Bluetooth range of my central Home Assistant Pi.



Kevin Sonney is a technology professional, media producer, and podcaster.

A Linux Sysadmin and Open Source advocate, Kevin has over 25 years in the IT industry, with over 15 years in Open Source. He currently works as an SRE at elastic.

Kevin hosts the weekly Productivity Alchemy Podcast. He and his wife, author and illustrator Ursula Vernon, co-host the weekly podcast Kevin and Ursula Eat Cheap (NSFW) and routinely attend sci-fi and comic conventions. Kevin also voiced Rev. Mord on The Hidden Almanac.

Kevin and Ursula live in Pittsboro, NC with a variety of dogs, cats, and chickens.

## 5 Rust tools worth trying on the Linux command line

By Sudeshna Sur

Linux inherited a lot from Unix, which has been around for a half-century. This means most of the tools you use in your Linux terminal probably either have a very long history or were written to emulate those historical commands. It's a point of pride in the POSIX world that tools don't *need* constant reinvention. In fact, there's a subset of Linux users today who could run a version of Linux from [before they were born](#) without having to learn anything new. It's tried, true, and reliable.

That doesn't mean there hasn't been evolution, though. All the commands Linux users know and love have been improved over the years. Some have even been replaced entirely and are so common now that few people still care to remember the old ones. Can you imagine Linux without SSH? Well, believe it or not, the `ssh` command replaced one called `rsh`.

I'm often on the lookout for new commands because I'm always intrigued by the possibility of getting things done more efficiently. If there's a better, faster, or more robust command out there for doing a common task, I want to know about it. And while there's equal opportunity for any language to invent new Linux commands, Rust developers have been delivering an impressive collection of useful general-purpose utilities.

## Replace man with tealdeer

Tealdeer provides the `tldr` command, which displays an abbreviated, no-nonsense summary of how a command is used. It's not that manual and info pages aren't useful, because they are, but sometimes they can be a little verbose and a little obtuse. Tealdeer keeps its hints clear and concise, with examples of how to use the command you're struggling to recall.

```
$ tldr tar
Archiving utility.
Often combined with a compression method, such as gzip or bzip2.
More information: <https://www.gnu.org/software/tar>.

[c]reate an archive and write it to a [f]ile:
    tar cf target.tar file1 file2 file3

[c]reate a g[z]ipped archive and write it to a [f]ile:
    tar czf target.tar.gz file1 file2 file3

[c]reate a g[z]ipped archive from a directory using relative paths:
    tar czf target.tar.gz --directory=path/to/directory .
[...]
```

Read the full article [about tldr](#).

## Replace du with dust

The `du` command gives feedback about disk usage. It's a relatively simple task; likewise, the command is pretty simple, too. The `dust` command is `du` written in Rust, and it uses color-coding and bar graphs for users who prefer added visual context.

```
$ dust
5.7M   ┌─ exa           |
█ | 2%                               |
5.9M   ┌─ tokei        |
█ | 2%                               |
6.1M   ┌─ dust         |
█ | 2%                               |
6.2M   ┌─ tldr         |
█ | 2%                               |
9.4M   ┌─ fd           |
█ | 4%                               |
2.9M   ┌─ exa         |
[...]
```

Read the full article [about dust](#).

## Replace find with fd

The `find` command is a useful tool for finding files on your computer, but its syntax can be difficult to master. Not only are there a lot of options, but the order of those options can be significant, depending on what you're doing. Some people have [written scripts](#) to abstract the task away from the command. Other people just write a new tool altogether, like `fd`.

Syntax doesn't get any easier than this:

```
$ fd example
Documents/example.txt
Documents/example-java
Downloads/example.com/index.html
```

Read the full article [about fd](#).

## Replace ls with exa

You might not think that the `ls` command would have much room for improvement. But `exa` proves that even the most mundane utility can benefit from small adjustments. For instance, why not have a list command with built-in Git awareness? Why not get extra metadata in your file lists?

Read the full [article about exa](#).

## Try Tokei

Unlike the other tools on this list, the `tokei` utility doesn't replace one command, but it does demonstrate how the Linux terminal is—as always—an environment very much in constant growth. The terminal may contain lots of legacy commands, but there are new and exciting commands surfacing all the time.

When I'm looking at a project in my local file system, and I need to know what languages it contains, I rely on a tool like Tokei. It's a program that displays statistics about a codebase, with wide support for 150 programming languages. I don't need to remember what languages have been used, or how many lines of code there are, or how many blanks or spaces or comments are there. It's a complete code-analysis tool, making my entry into and navigation of the code easy.

```
$ tokei ~/exa/src ~/Work/wildfly/jaxrs
```



Language	Files	Lines	Code	Comments	Blank
Java	46	6135	4324	945	632
XML	23	5211	4839	473	224
-----					
Rust					
Markdown					
-----					
Total					

Read the full [article about tokei](#).

## Find your favorite

Open source users never have to settle for just a small set of commands, or even just one version of a command. Find the commands you love, whether they're new ideas for emerging workflows, or reimplementations of old tools, or timeless classics that are just as good today as they were decades ago. Find the commands that make your life better and enjoy!



Sudeshna is from Kolkata and currently working for the Red Hat Middleware team in Pune, India. She loves to explore different open source projects and programs. She started contributing to Open Source a couple of years back as an Hacktober Participant since then she has been an avid blogger at Dzone on Python and Data Science. In her free time, she enjoys stargazing and sampling continental cuisine. She is an RHCSA and is working her way to RHCA.

## 4 Linux tools to erase your data

By Don Watkins

One of the best ways to keep your data secure is by only writing data to an encrypted hard drive. On a standard drive, it's possible to view data just by mounting the drive as if it were a thumb drive, and it's even possible to display and recover even deleted data with tools like [Scalpel](#) and [Testdisk](#). But on an encrypted drive, data is unreadable without a decryption key (usually a passphrase you enter when mounting the drive.)

Encryption can be established when you install your OS, and some operating systems even make it possible to activate encryption any time after installation.

What do you do when you're selling a computer or replacing a drive that never got encrypted in the first place, though?

The next best thing to encrypting your data from the start is by erasing the data when you're finished with the drive.

### Responsible caretaker

I'm frequently called on to help clients upgrade an old computer. Invariably, they're more than willing to help me recycle them so that they can be used by someone else. I'm happy to refurbish these older computers and refit them with a newer solid-state drive, dramatically improving performance.

However, it's not a good idea to just throw an old drive in the trash. It needs to be erased and then disposed of properly. Rather than leave the drives in the original computer, I remove them, place them in a drive enclosure, and connect them to my Linux computer. Several Linux utilities can easily accomplish this. One of them is the **Gnu Shred** tool.

## GNU Shred

```
$ sudo shred -vzf /dev/sdX
```

Shred has many options:

- **n** - the number of overwrites. The default is three.
- **u** - overwrite and delete.
- **s** - the number of bytes to shred.
- **v** - show extended information.
- **f** - force the change of permissions to allow writing if necessary.
- **z** - add a final overwrite with zeros to hide shredding.

Use `shred --help` for more information

## ShredOS

ShredOS is a live Linux distribution with the sole purpose of erasing the entire contents of a drive. It was developed after a similar distribution, called DBAN, was discontinued. It uses the `nwipe` application, which is a fork of DBAN's `dwipe`. You can make a bootable USB drive by downloading the 32 bit or 64 bit image and writing it to a drive with the `dd` command on Linux and macOS:

```
$ sudo dd if=shredos.img of=/dev/sdX bs=4M status=progress
```

Alternately, you can use the [Etcher](#) tool on Linux, macOS, and Windows.

## The dd command

A common method for erasing drives is with the Linux `dd` command. Nearly every Linux installation comes with the `dd` utility installed. Make sure that the drive is not mounted.

```
$ sudo umount /dev/sdXY -l
```

If you want to write zeros over your entire target disk, issue the following command. It will probably be an overnight job.

```
$ sudo dd if=/dev/urandom of=/dev/sdX bs=10M
```

**Warning:** Be sure that you know where you are on your system and target the correct drive so that you don't accidentally erase your own data.

## Nvme-cli

If your computer contains one of the newer NVMe drives, you can install the [nvme-cli](#) utilities and use the `sanitize` option to erase your drive.

The command `nvme sanitize help` command provides you with a list of sanitize options, which include the following:

- `--no-dealloc, -d` - No deallocate after sanitize.
- `--oipbp, -i` - Overwrite invert pattern between passes.
- `--owpass=, -n` - Overwrite pass count.
- `--ause, -u` - Allow unrestricted sanitize exit.
- `--sanact=, -a` - Sanitize action.
- `--ovrpat=, -p` - Overwrite pattern.

Here is the command I use:

```
$ sudo nvme sanitize /dev/nvme0nX
```

The same warnings apply here as with the format process: back up important data first because this command erases it!

## Information management

The information you keep on your computer is important. It belongs to you and to know one else. When you're selling off a computer or disposing of a hard drive, make sure you've cleared it of your data with one of these great tools.



Educator, entrepreneur, open source advocate, life long learner, Python teacher. M.A. in Educational Psychology, MSED in Educational Leadership, Linux system administrator.

Follow Don on Twitter at [@Don\\_Watkins](#)

# 5 agile mistakes I've made and how to solve them

By Kelsea Zhang

[Agile](#) used to have a stigma as being "only suitable for small teams and small project management." It is now a famous discipline used by software development teams worldwide with great success. But does agile really deliver value? Well, it depends on how you use it.

My teams and I have used agile since I started in tech. It hasn't always been easy, and there's been a lot of learning along the way. The best way to learn is to make mistakes, so to help you in your own agile journey, here are five agile mistakes I've made.

## 1. Mistake: Agile only happens in development teams

Here's what happens when you restrict agile to just your development team. Your business team writes requirements for a project, and that goes to the development team, with a deadline. In this case, the development team isn't directly responsible for business goals.

There's very little communication between teams, let alone negotiation. No one questions the demands made by the business team, or whether there's a better way to meet the same business goal.

This can be discouraging to development teams, too. When developers are only responsible for filling in the code to make the machine work, they're disconnected from the business.

The final product becomes a monster, lacking reasonable abstraction and design.

**Solution:** Spread agile through your organization. Let everyone benefit from it in whatever way that's appropriate for their department, but most importantly let it unify everyone's goals.

## 2. Mistake: Automated testing is too much work to setup

The role of automated testing, especially Test Driven Development (TDD), is often undervalued by the IT industry. In my opinion, automated testing is the cornerstone of maintainable and high-quality software, and is even more important than production code.

However, most teams today don't have the ability to automate testing, or have the ability but refuse it because of time constraints. Programmers lack the ability to continuously refactor bad code without the protection of automated testing.

This is because no one can predict whether changing a few lines of code will cause new bugs. Without continuous refactoring, you increase your technical debt, which reduces your responsiveness to the demands of your business units.

Manual testing is slow, and forces you to sacrifice quality, testing just the changed part (which can be difficult), or lengthening the regression testing time. If the test time is too long, you have to test in batches to reduce the number of tests performed.

Suddenly, you're not agile any more. You've converted to Waterfall.

**Solution:** The key to automated testing is to have developers run tests, instead of hiring more testers to write scripts. That's why tests (written by testers) run slowly and only slowly produce feedback to programmers.

What's needed to improve code quality is rapid feedback on the program. The earlier an automated test is written, and the faster it's run, the more conducive it is for programmers to get feedback in a timely manner.

The fastest way to write automated tests is [TDD](#). Write tests before you write the production code. The fastest way to run automated tests is unit testing.

## 3. Mistake: As long as it works, you can ignore code quality

People often say, "We're running out of time, just finish it."

They don't care about quality. Many people think that quality can be sacrificed for efficiency. So you end up writing low-quality code because you do not have time for anything else. In addition, low-quality code doesn't result in high performance.

Unless your program is as simple as a few lines of code, low-quality code will hold you back as code complexity increases. Software is called "soft" because we expect it to be easy to change. Low-quality code becomes increasingly difficult to change because a small change can lead to thousands of new bugs.

**Solution:** The only way to improve code quality is to improve your skills. Most people can't write high-quality code in one sitting. That's why you need constant refactoring! (And you must implement automated testing to support constant refactoring).

## 4. Mistake: Employees should specialize in just one thing

It feels natural to divide personnel into specialized teams. One employee might belong to the Android group, another to the iOS group, another to the background group, and so on. The danger is that teams with frequent changes mean that specialization is difficult to sustain.

**Solution:** Many practices in agile are based on teams such as team velocity, retrospective improvement, and staff turnover. Agile practices revolve around teams and around people. Help your team members diversify, learn new skills, and share knowledge.

## 5. Mistake: Writing requirements takes too much time

As the saying goes "Garbage in Garbage out," and a formal software requirement is the "input" of software development. Good software cannot be produced without clear requirements.

In the tech industry, I have found that good product owners are more scarce than good programmers. After all, no matter how poorly a programmer writes code, it usually at least runs (or else it doesn't ship).

For most product managers, there is no standard to measure the efficacy of their product definitions and requirements. Here are a few of the issues I've seen over the years:

- Some product owners are devoted to designing solutions while ignoring user value.

This results in a bunch of costly, but useless functions.

- Some product managers can only tell big stories, and can't split requirements into small, manageable pieces, resulting in large delivery batches and reduced agility.
- Some product owners have incomplete requirement analysis, resulting in bug after bug.

- Sometimes product owners don't prioritize requirements, which leads to teams wasting a lot of time on low-value items.

**Solution:** Create clear, concise, and manageable requirements to help guide development.



## Make mistakes

I've given you five tips on some mistakes to avoid. Don't worry, though, there are still plenty of mistakes to make! Take agile to your organization, don't be afraid of enduring a few mistakes for the benefit of making your teams better.

Once you've taken the inevitable missteps, you'll know what to do differently the next time around. Agility is designed to survive mistakes. That's one of its strengths: it can adapt. So get started with agile, be ready to adapt, and make better software!

---



Kelsea Zhang is one of the members of [ZenTao](#) team. ZenTao is a project management tool that covers the entire software development lifecycle, which has [won first place in the "Most Used Test Management Tools" for seven years in a row](#). Kelsea Zhang is a bicycle enthusiast, a jogger, and a writer.

# Run containers on Mac with Lima

By Moshe Zadka

Running containers on your Mac can be a challenge. After all, containers are based on Linux-specific technologies like cgroups and namespaces. However, macOS has a built-in hypervisor, allowing virtual machines (VMs) on the Mac. The hypervisor is a low-level kernel feature, not a user-facing one.

Enter `hyperkit`, an [open source project](#) that runs VMs using the macOS hypervisor. The `hyperkit` tool is designed to be a "minimalist" VM runner. Unlike VirtualBox, it doesn't come with fancy UI features to manage VMs.

You can grab `hyperkit`, a minimalist Linux distribution running a container manager, and plumb all the pieces together. That's a lot of moving parts and is a lot of work, especially to make network connections seamless by using `vpnkit`, an open source project to create a VM's network that feels more like part of the host's network.

## Lima

There is no reason to go to all that effort, when [the Lima project](#) has figured out the details. One of the easiest ways to get `lima` running is with [Homebrew](#). You can install `lima` with this command:

```
$ brew install lima
```

After installation, which might take a while, it is time to begin having some fun. In order to let `lima` know you are ready for some fun, you need to start it. Here's the command:

```
$ limactl start
```

If this is your first time, you're asked whether you like the defaults or whether you want to change any of them. The defaults are pretty safe, but I like to live on the wild side. This is why I jump into an editor and make the following modifications from:

```
- location: "~"
  # CAUTION: `writable` SHOULD be false for the home directory.
  # Setting `writable` to true is possible but untested and dangerous.
  writable: false
```

to:

```
- location: "~"
  # I *also* like to live dangerously -- Austin Powers
  writable: true
```

As it says in the comment, this can be dangerous. Many existing workflows, sadly, depend on this mounting to be read-write.

By default, `lima` runs `containerd` to manage containers. The `containerd` manager is also a pretty frill-less one. While it is not uncommon to use a wrapper daemon, like `dockerd`, to add those nice-to-have ergonomics, there is another way.

## The `nerdctl` tool

The `nerdctl` tool is a drop-in replacement for the Docker client which puts those features in the client, not the server. The `lima` tool allows running `nerdctl` without installing it locally, directly from inside the VM.

Putting it all together, it is time to run a container! This container will run an HTTP server. You can create the files on your Mac:

```
$ ls
index.html
$ cat index.html
hello
```

Now, mount and forward the ports:

```
$ lima nerdctl run --rm -it -p 8000:8000 -v $(pwd):/html --entrypoint bash python
root@9486145449ab:/#
```

Inside the container, run a simple web server:

```
$ lima nerdctl run --rm -it -p 8000:8000 -v $(pwd):/html --entrypoint bash python
root@9486145449ab:/# cd /html/
root@9486145449ab:/html# python -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

From a different terminal, you can check that everything looks good:

```
$ curl localhost:8000
hello
```

Back on the container, there is a log message documenting the HTTP client's connection:

```
10.4.0.1 - - [09/Sep/2021 14:59:08] "GET / HTTP/1.1" 200 -
```

One file is not enough. Stop the server with **CTRL-C** and add another file:

```
^C
Keyboard interrupt received, exiting.
root@9486145449ab:/html# echo goodbye > foo.html
root@9486145449ab:/html# python -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Check that you can see the new file:

```
$ curl localhost:8000/foo.html
goodbye
```

Installing `lima` takes a while, but after you are done, you can run containers, mount arbitrary sub-directories of your home directory into containers, edit files in those directories, and run network servers that appear to macOS programs like they're running on localhost. All with `lima nerdctl`.



Moshe has been involved in the Linux community since 1998. He's been programming Python since 1999, and has contributed to the core Python interpreter. Moshe has been a DevOps/SRE since before those terms existed, caring deeply about software reliability, build reproducibility and other such things.