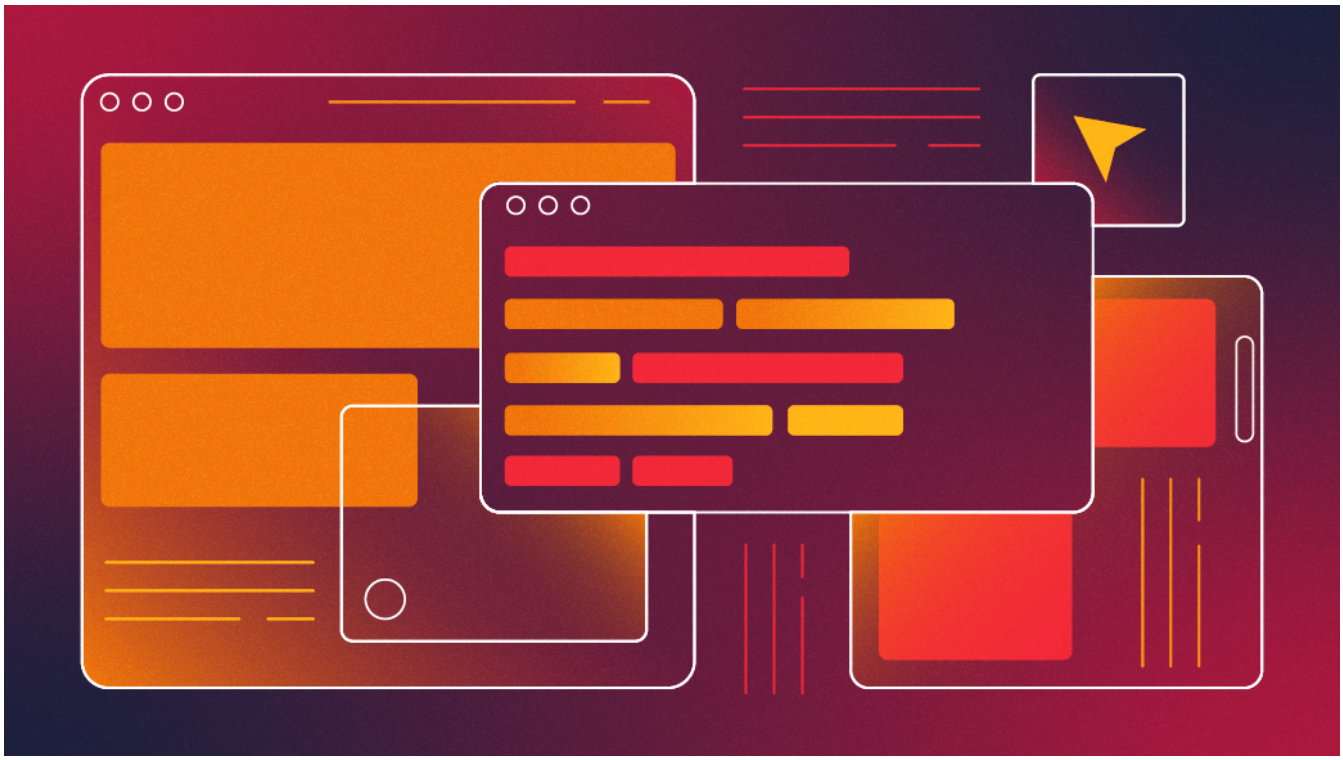
Host your website with dynamic content and a database on a Raspberry Pi

You can use free software to support a web application on a very lightweight computer.

By Marty Kalin

March 3, 2021 | 1 Comment | 15 min read

159 readers like this.



Raspberry Pi's single-board machines have set the mark for cheap, real-world computing. With its model 4, the Raspberry Pi can host web applications with a production-grade web server, a transactional database system, and dynamic content through scripting. This article explains the installation and configuration

details with a full code example. Welcome to web applications hosted on a very lightweight computer.

The snowfall application

More on Raspberry Pi

What is Raspberry Pi?

eBook: Guide to Raspberry Pi

Getting started with Raspberry Pi cheat sheet

eBook: Running Kubernetes on your Raspberry Pi

Whitepaper: Data-intensive intelligent applications in a hybrid cloud blueprint

Understanding edge computing

Our latest on Raspberry Pi

Imagine a downhill ski area large enough to have microclimates, which can mean dramatically different snowfalls across the area. The area is divided into regions, each of which has devices that record snowfall in centimeters; the recorded information then guides decisions on snowmaking, grooming, and other maintenance operations. The devices communicate, say, every 20 minutes with a server that updates a database that supports reports. Nowadays, the server-side software for such an application can be free *and* production-grade.

This snowfall application uses the following technologies:

A Raspberry Pi 4 running Debian

Nginx web server: The free version hosts over 400 million websites. This web server is easy to install, configure, and use.

SQLite relational database system, which is file-based: A database, which can hold many tables, is a file on the local system. SQLite is lightweight but also ACID-compliant; it is suited for low to moderate volume. SQLite is likely the most widely used database system in the world, and the source code for

SQLite is in the public domain. The current version is 3. A more powerful (but still free) option is PostgreSQL.

Python: The Python programming language can interact with databases such as SQLite and web servers such as Nginx. Python (version 3) comes with Linux and macOS systems.

Python includes a software driver for communicating with SQLite. There are options for connecting Python scripts with Nginx and other web servers. One option is uWSGI (Web Server Gateway Interface), which updates the ancient CGI (Common Gateway Interface) from the 1990s.

Several factors speak for uWSGI:

uWSGI is flexible. It can be used as either a lightweight concurrent web server or the backend application server connected to a web server such as Nginx.

Its setup is minimal.

The snowfall application involves a low to moderate volume of hits on the web server and database system. In general, CGI technologies are not fast by modern standards, but CGI performs well enough for department-level web applications such as this one.

Various acronyms describe the uWSGI option. Here's a sketch of the three principal ones:

WSGI is a Python specification for an interface between a web server on one side, and an application or an application framework (e.g., Django) on the other side. This specification defines an API whose implementation is left open.

uWSGI implements the WSGI interface by providing an application server, which connects applications to a web server. A uWSGI application server's main job is to translate HTTP requests into a format that a web application can consume and, afterward, to format the application's response into an HTTP message.

uwsgi is a binary protocol implemented by a uWSGI application server to communicate with a full-featured web server such as Nginx; it also includes utilities such as a lightweight web server. The Nginx web server "speaks" uwsgi out of the box.

For convenience, I will use "uwsgi" as shorthand for the binary protocol, the application server, and the very lightweight web server.

Setting up the database

On a Debian-based system, you can install SQLite the usual way (with `%` representing the command-line prompt):

```
% sudo apt-get install sqlite3
```

This database system is a collection of C libraries and utilities, all of which come to about 500KB in size. There is no database server to start, stop, or otherwise maintain.

Once SQLite is installed, create a database at the command-line prompt:

```
% sqlite3 snowfall.db
```

If this succeeds, the command creates the file `snowfall.db` in the current working directory. The database name is arbitrary (e.g., no extension is required), and the command opens the SQLite client utility with `>sqlite` as the prompt:

```
Enter ".help" for usage hints.
sqlite>
```

Create the snowfall table in the snowfall database with the following command. The table name, like the database name, is arbitrary:

```
sqlite> CREATE TABLE snowfall (id INTEGER PRIMARY KEY AUTOIN
                               region TEXT NOT NULL,
                               device TEXT NOT NULL,
                               amount DECIMAL NOT NULL,
                               tstamp DECIMAL NOT NULL);
```

SQLite commands are case-insensitive, but it is traditional to use uppercase for SQL terms and lowercase for user terms. Check that the table was created:

```
sqlite> .schema
```

The command echoes the `CREATE TABLE` statement.

The database is now ready for business, although the single-table snowfall is empty. You can add rows interactively to the table, but an empty table is fine for now.

A first look at the overall architecture

Recall that uwsgi can be used in two ways: either as a lightweight web server or as an application server connected to a production-grade web server such as Nginx. The second use is the goal, but the first is suited for developing and testing the programmer's request-handling code. Here's the architecture with Nginx in play as the web server:

```
          HTTP       uwsgi
client<---->Nginx<----->appServer<--->request-handling code<--->SQ
```

The client could be a browser, a utility such as curl, or a hand-crafted program fluent in HTTP. Communications between the client and Nginx occur through HTTP, but then uwsgi takes over as a binary-transport protocol between Nginx and the application server, which interacts with request-handling code such as `requestHandler.py` (described below). This architecture delivers a clean division of labor. Nginx alone manages the client, and only the request-handling code interacts with the database. In turn, the application server separates the web server from the programmer-written code, which has a high-level API to read and write HTTP messages delivered over uwsgi.

I'll examine these architectural pieces and cover the steps for installing, configuring, and using uwsgi and Nginx in the next sections.

The snowfall application code

Below is the source code file `requestHandler.py` for the snowfall application. (It's also available on my website.) Different functions within this code help clarify the software architecture that connects SQLite, Nginx, and uwsgi.

The request-handling program

```
import sqlite3
import cgi

PATH_2_DB = '/home/marty/wsgi/snowfall.db'

## Dispatches HTTP requests to the appropriate handler.
def application(env, start_line):
    if env['REQUEST_METHOD'] == 'POST':   ## add new DB reco
        return handle_post(env, start_line)
    elif env['REQUEST_METHOD'] == 'GET':  ## create HTML-fra
        return handle_get(start_line)
    else:                                 ## no other option
        start_line('405 METHOD NOT ALLOWED', [('Content-Type
        response_body = 'Only POST and GET verbs supported.'
        return [response_body.encode()]

def handle_post(env, start_line):
    form = get_field_storage(env)  ## body of an HTTP POST r

    ## Extract fields from POST form.
    region = form.getvalue('region')
    device = form.getvalue('device')
    amount = form.getvalue('amount')
    tstamp = form.getvalue('tstamp')

    ## Missing info?
    if (region is not None and
        device is not None and
        amount is not None and
        tstamp is not None):
        add_record(region, device, amount, tstamp)
        response_body = "POST request handled.\n"
        start_line('201 OK', [('Content-Type', 'text/plain')
    else:
        response_body = "Missing info in POST request.\n"
        start_line('400 Bad Request', [('Content-Type', 'tex

    return [response_body.encode()]

def handle_get(start_line):
    conn = sqlite3.connect(PATH_2_DB)        ## connect to D
```

```
    cursor = conn.cursor()                  ## get a cursor
    cursor.execute("select * from snowfall")

    response_body = "<h3>Snowfall report</h3><ul>"
    rows = cursor.fetchall()
    for row in rows:
        response_body += "<li>" + str(row[0]) + '|'   ## prim
        response_body += row[1] + '|'                 ## regi
        response_body += row[2] + '|'                 ## devi
        response_body += str(row[3]) + '|'            ## amou
        response_body += str(row[4]) + "</li>"        ## time
    response_body += "</ul>"

    conn.commit()  ## commit
    conn.close()   ## cleanup

    start_line('200 OK', [('Content-Type', 'text/html')])
    return [response_body.encode()]

## Add a record from a device to the DB.
def add_record(reg, dev, amt, tstamp):
    conn = sqlite3.connect(PATH_2_DB)      ## connect to DB
    cursor = conn.cursor()                 ## get a cursor

    sql = "INSERT INTO snowfall(region,device,amount,tstamp)
    cursor.execute(sql, (reg, dev, amt, tstamp)) ## execute

    conn.commit()  ## commit
    conn.close()   ## cleanup

def get_field_storage(env):
    input = env['wsgi.input']
    form = env.get('wsgi.post_form')
    if (form is not None and form[0] is input):
        return form[2]

    fs = cgi.FieldStorage(fp = input,
                          environ = env,
                          keep_blank_values = 1)
    return fs
```

A constant at the start of the source file defines the path to the database file:

```
PATH_2_DB = '/home/marty/wsgi/snowfall.db'
```

Make sure to update the path for your Raspberry Pi.

As noted earlier, uwsgi includes a lightweight web server that can host this request-handling application. To begin, install uwsgi with these two commands (## introduces my comments):

```
% sudo apt-get install build-essential python-dev ## C heade
% pip install uwsgi                                ## pip = P
```

Next, launch a bare-bones snowfall application using uwsgi as the web server:

```
% uwsgi --http 127.0.0.1:9999 --wsgi-file requestHandler.py
```

The flag `--http` runs uwsgi in web-server mode, with 9999 as the web server's listening port on localhost (127.0.0.1). By default, uwsgi dispatches HTTP requests to a programmer-defined function named `application`. For review, here's the full function from the top of the `requestHandler.py` code:

```
def application(env, start_line):
    if env['REQUEST_METHOD'] == 'POST':   ## add new DB reco
        return handle_post(env, start_line)
    elif env['REQUEST_METHOD'] == 'GET':  ## create HTML-fra
        return handle_get(start_line)
    else:                                 ## no other option
        start_line('405 METHOD NOT ALLOWED', [('Content-Type
        response_body = 'Only POST and GET verbs supported.'
        return [response_body.encode()]
```

The snowfall application accepts only two request types:

> A POST request, if up to snuff, creates a new entry in the snowfall table. The request should include the ski area region, the device in the region, the snowfall amount in centimeters, and a Unix-style timestamp. A POST request is dispatched to the `handle_post` function (which I'll clarify shortly).

A GET request returns an HTML fragment (an unordered list) with the records currently in the snowfall table.

Requests with an HTTP verb other than POST and GET will generate an error message.

You can use a utility such as curl to generate HTTP requests for testing. Here are three sample POST requests to start populating the database:

```
% curl -X POST -d "region=R1&device=D9&amount=1.42&tstamp=16
% curl -X POST -d "region=R7&device=D4&amount=2.11&tstamp=16
% curl -X POST -d "region=R5&device=D1&amount=1.12&tstamp=16
```

These commands add three records to the snowfall table. A subsequent GET request from curl or a browser displays an HTML fragment that lists the rows in the snowfall table. Here's the equivalent as non-HTML text:

```
Snowfall report

    1|R1|D9|1.42|1604722088.0158753
    2|R7|D4|2.11|1604722296.8862638
    3|R5|D1|1.12|1604942236.1013834
```

A professional report would convert the numeric timestamps into human-readable ones. But the emphasis, for now, is on the architectural components in the snowfall application, not on the user interface.

The uwsgi utility accepts various flags, which can be given either through a configuration file or in the launch command. For example, here's a richer launch of uwsgi as a web server:

```
% uwsgi --master --processes 2 --http 127.0.0.1:9999 --wsgi-
```

This version creates a master (supervisory) process and two worker processes, which can handle the HTTP requests concurrently.

In the snowfall application, the functions `handle_post` and `handle_get` process POST and GET requests, respectively. Here's the `handle_post` function in full:

```
def handle_post(env, start_line):
    form = get_field_storage(env)  ## body of an HTTP POST r

    ## Extract fields from POST form.
    region = form.getvalue('region')
    device = form.getvalue('device')
    amount = form.getvalue('amount')
    tstamp = form.getvalue('tstamp')

    ## Missing info?
    if (region is not None and
        device is not None and
        amount is not None and
        tstamp is not None):
        add_record(region, device, amount, tstamp)
        response_body = "POST request handled.\n"
        start_line('201 OK', [('Content-Type', 'text/plain')
    else:
        response_body = "Missing info in POST request.\n"
        start_line('400 Bad Request', [('Content-Type', 'tex

    return [response_body.encode()]
```

The two arguments to the `handle_post` function (`env` and `start_line`) represent the system environment and a communications channel, respectively. The `start_line` channel sends the HTTP start line (in this case, either `400 Bad Request` or `201 OK`) and any HTTP headers (in this case, just `Content-Type: text/plain`) of an HTTP response.

The `handle_post` function tries to extract the relevant data from the HTTP POST request and, if it's successful, calls the function `add_record` to add another row to the snowfall table:

```
def add_record(reg, dev, amt, tstamp):
    conn = sqlite3.connect(PATH_2_DB)      ## connect to DB
    cursor = conn.cursor()                 ## get a cursor

    sql = "INSERT INTO snowfall(region,device,amount,tstamp)
    cursor.execute(sql, (reg, dev, amt, tstamp)) ## execute
```

```
    conn.commit()  ## commit
    conn.close()   ## cleanup
```

SQLite automatically wraps single SQL statements (such as `INSERT` above) in a transaction, which accounts for the call to `conn.commit()` in the code. SQLite also supports multi-statement transactions. After calling `add_record`, the `handle_post` function winds up its work by sending an HTTP response confirmation message to the requester.

The `handle_get` function also touches the database, but only to read the records in the snowfall table:

```
def handle_get(start_line):
    conn = sqlite3.connect(PATH_2_DB)        ## connect to D
    cursor = conn.cursor()                   ## get a cursor
    cursor.execute("SELECT * FROM snowfall")

    response_body = "<h3>Snowfall report</h3><ul>"
    rows = cursor.fetchall()
    for row in rows:
        response_body += "<li>" + str(row[0]) + '|'  ## prim
        response_body += row[1] + '|'                ## regi
        response_body += row[2] + '|'                ## devi
        response_body += str(row[3]) + '|'           ## amou
        response_body += str(row[4]) + "</li>"       ## time
    response_body += "</ul>"

    conn.commit()  ## commit
    conn.close()   ## cleanup

    start_line('200 OK', [('Content-Type', 'text/html')])
    return [response_body.encode()]
```

A user-friendly version of the snowfall application would support additional (and fancier) reports, but even this version of `handle_get` underscores the clean interface between Python and SQLite. By the way, uwsgi expects a response body to be a list of bytes. In the `return` statement, the call to `response_body.encode()` inside the square brackets generates the byte list from the `response_body` string.

Moving up to Nginx

The Nginx web server can be installed on a Debian-based system with one command:

```
% sudo apt-get install nginx
```

As a web server, Nginx provides the expected services, such as wire-level security, HTTPS, user authentication, load balancing, media streaming, response compression, file uploading, etc. The Nginx engine is high-performance and stable, and this server can support dynamic content through a variety of programming languages. Using uwsgi as a very lightweight web server is an attractive option but switching to Nginx is a move up to industrial-strength web hosting with high-volume capability. Nginx and uwsgi are both implemented in C.

With Nginx in play, uwsgi takes on a communication protocol's restricted roles and an application server; it no longer acts as an HTTP web server. Here's the revised architecture:

```
           HTTP       uwsgi
requester<---->Nginx<----->app server<--->requestHandler.py
```

As noted earlier, Nginx includes uwsgi support and now acts as a reverse-proxy server that forwards designated HTTP requests to the uwsgi application server, which in turn interacts with the Python script `requestHandler.py`. Responses from the Python script move in the reverse direction so that Nginx sends the HTTP response back to the requesting client.

Two changes bring this new architecture to life. The first launches uwsgi as an application server:

```
% uwsgi --socket 127.0.0.1:8001 --wsgi-file requestHandler.p
```

Socket 8001 is the Nginx default for uwsgi communications. For robustness, you could use the full path to the Python script so that the command above does not have to be executed in the directory that houses the Python script. In a production

environment, uwsgi would start and stop automatically; for now, however, the emphasis remains on how the architectural pieces fit together.

The second change involves Nginx configuration, which can be tricky on Debian-based systems. The main configuration file for Nginx is `/etc/nginx/nginx.conf`, but this file may have `include` directives for other files, in particular, files in one of three `/etc/nginx` subdirectories: `nginx.d`, `sites-available`, and `sites-enabled`. The `include` directives can be eliminated to simplify matters; in this case, the configuration occurs only in `nginx.conf`. I recommend the simple approach.

However the configuration is distributed, the key section for having Nginx talk to the uwsgi application server begins with `http` and has one or more `server` subsections, which in turn have `location` subsections. Here's an example from the Nginx documentation:

```
...
http {
    # Configuration specific to HTTP and affecting all virtu
    ...
    server { # simple reverse-proxy
       listen       80;
       server_name  domain2.com www.domain2.com;
       access_log   logs/domain2.access.log  main;

       # serve static files
       location ~ ^/(images|javascript|js|css|flash|media|st
         root    /var/www/virtual/big.server.com/htdocs;
         expires 30d;
       }

       # pass requests for dynamic content to rails/turbogea
       location / {
         proxy_pass      http://127.0.0.1:8080;
       }
     }
     ...
}
```

The `location` subsections are the ones of interest. For the snowfall application, here's the added `location` entry with its two configuration lines:

```
...
server {
   listen 80 default_server;
   listen [::]:80 default_server;

   root /var/www/html;
   index index.html index.htm index.nginx-debian.html;

   server_name _;

   ### key addition for uwsgi communication
   location /snowfall {
      include uwsgi_params;       ## comes with Nginx
      uwsgi_pass 127.0.0.1:8001;  ## 8001 is the default for
   }
   ...
}
...
```

To keep things simple for now, make `/snowfall` the only `location` in the configuration. With this configuration in place, Nginx listens on port 80 and dispatches HTTP requests ending with the `/snowfall` path to the uwsgi application server:

```
% curl -X POST -d "..." localhost/snowfall ## new POST
% curl -X GET localhost/snowfall           ## new GET
```

The port number 80 can be dropped from the request because 80 is the default server port for HTTP requests.

If the configured location were simply `/` instead of `/snowfall`, then any HTTP request with `/` at the start of the path would be dispatched to the uwsgi application server. Accordingly, the `/snowfall` path leaves room for other locations and, therefore, for further actions in response to HTTP requests.

Once you've changed the Nginx configuration with the added `location` subsection, you can start the web server:

```
% sudo systemctl start nginx
```

There are other commands similar to `stop` and `restart` Nginx. In a production environment, you could automate these actions so that Nginx starts on a system boot and stops on a system shutdown.

With uwsgi and Nginx both running, you can use a browser to test whether the architectural components cooperate as expected. For example, if you enter the URL `localhost/` in the browser's input window, then the Nginx welcome page should appear with (HTML) content similar to this:

```
Welcome to nginx!
...
Thank you for using nginx.
```

By contrast, the URL `localhost/snowfall` should display the rows currently in the snowfall table:

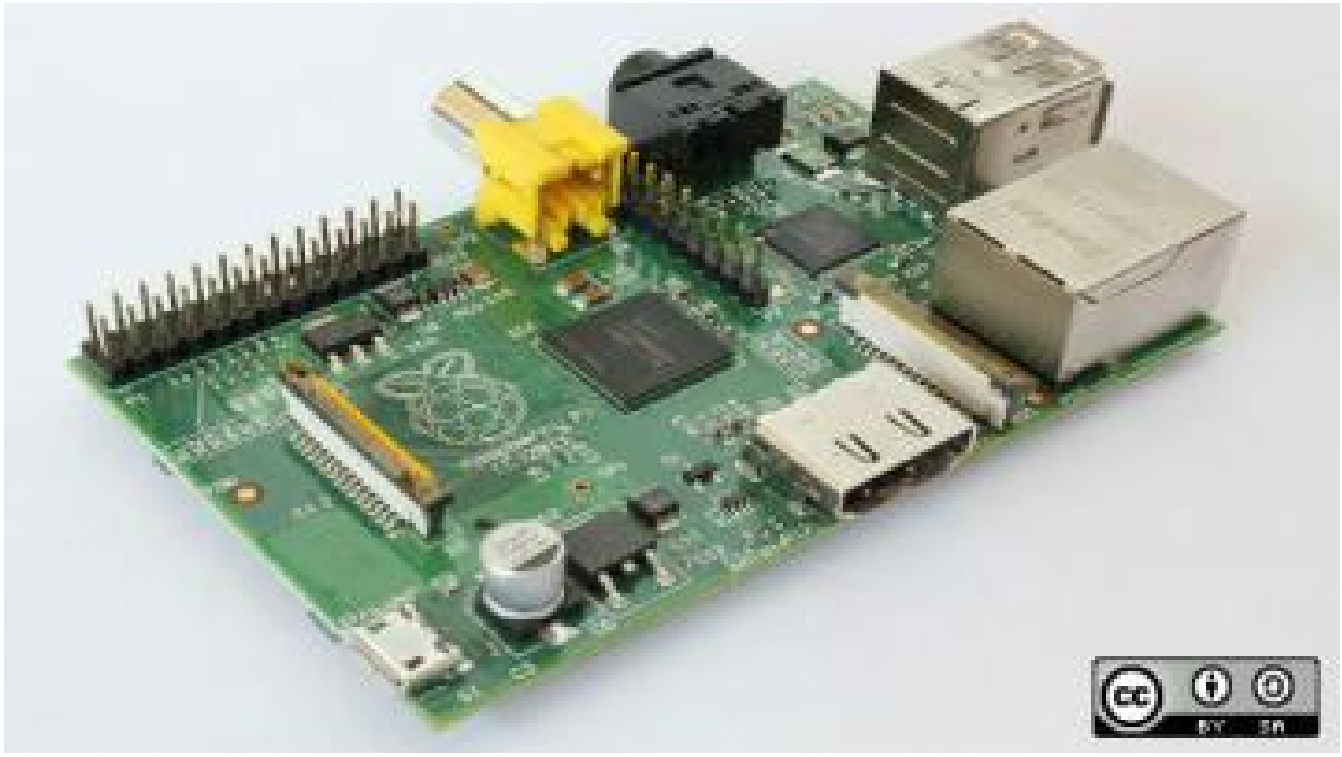
```
Snowfall report

    1|R1|D9|1.42|1604722088.0158753
    2|R7|D4|2.11|1604722296.8862638
    3|R5|D1|1.12|1604942236.1013834
```

Wrapping up

The snowfall application shows how free software components—a high-powered web server, an ACID-compliant database system, and scripting for dynamic content—can support a realistic web application on a Raspberry Pi 4 platform. This lightweight machine lifts above its weight class, and Debian eases the lifting.

The software components in the web application work well together and require very little configuration. For higher volume hits against a relational database, recall that a free and feature-rich alternative to SQLite is PostgreSQL. If you're eager to play on the Raspberry Pi 4—in particular, to explore server-side web programming on this platform—then Nginx, SQLite or PostgreSQL, uwsgi, and Python are worth considering.

What to read next



Set up a minimal server on a Raspberry Pi

Don't decommission that old Raspberry Pi just yet! This step-by-step guide shows how I set up my Raspberry Pi with the most minimal configuration to conserve precious system...

Alan Formy-Duval (Correspondent)

Build a private social network with a Raspberry Pi

Step-by-step instructions on how to create your own social network with low-cost hardware and simple setup.

Giuseppe Cassibba

Tags: RASPBERRY PI

Marty Kalin

I'm an academic in computer science (College of Computing and Digital Media, DePaul University) with wide experience in software development, mostly in production planning and scheduling (steel industry) and product configuration (truck and bus manufacturing). Details on books and other publications are available at

More about me

1 Comment

These comments are closed, however you can Register or Login to post a comment on another article.

madtom1999 | March 5, 2021

No readers like this yet.

I've been hosting stuff on a Pi Zero W on our home/office using MariaDB, Apache and PHP (I've written quite a bit of commercial code in it) and it can easily handle a dozen clients or so for simple stuff. Not tried stress testing it 'properly' though.

Related Content

5 Raspberry Pi projects to do with this open source data tool

Measure pi with a Raspberry Pi

Control your Raspberry Pi with Lua

ABOUT THIS SITE

The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.

Opensource.com aspires to publish all content under a **Creative Commons license** but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.

A note on advertising: Opensource.com does not sell advertising on the site or in any of its newsletters.