Introduction

Welcome to Module 12! This week, we will learn about application performance. We will use profiling tools to identify performance bottlenecks within an application. Once bottlenecks are identified, optimization strategies will be employed to improve performance in your application.

Topics and Learning Objectives

Topics

- Performance bottlenecks, performance optimization
- Profiling, profiling tools
- Werkzeug, Flask-Debug Toolbar, Flask Profiler
- API Monitoring

Learning Objectives

By the end of the module, you should be able to:

1. Identify pros and cons of premature optimization and decide when to optimize and when not to optimize.
2. Find the highest leverage places to optimize.
3. Identify bottlenecks using profiling tools such as Werkzeug, Flask-Debug Toolbar and Flask Profiler.
4. Examine potential bottlenecks and strategies for optimizing them in order to provide a production application that executes efficiently.

Readings & Resources

Reading

Required

- Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python* (2nd ed.). O'Reilly Media, Inc.
  - Chapter 16: Performance

Video

- Mike Müller – Faster Python programs – Measure, don't guess – PyCon 2019 [3:18:00]
  - Note: Students are not required to watch the entire video; only watch clips pertaining to the project undertaken
- Ned Batchelder – Big-O: How code slows as data grows – PyCon 2018 [28:50]
- Matt Davis – Python performance investigation by example – PyCon 2018 [30:35]

Identify Bottlenecks

- Performance optimization is a difficult task to achieve. The major question to ask is where are the bottlenecks in our application?
- Identifying bottlenecks requires profiling or the understanding of execution times of various components in the application.
- What are components in an application: routing, business logic, database access, and so on?
- Why is a component slow? Is it due to CPU power (multi-core availability), shortage of memory, slow I/O (disk access) or a slow network connection?

Enabling Debug Mode in Python

- Debugging can provide information about how a Python/Flask program behaves.
- The code below enables debugging.

```
if __name__ == '__main__':
     app.run(debug=True)
```

- Enabling debugging requires overhead with respect to program execution. The overhead is in terms of CPU and memory usage.
- For production purposes it is recommended that debugging be turned off. That is setting the debug flag to false. This removes monitoring subroutines from your application. The end result is a faster application.

Optimizing Database Access

- SQL queries execute against a database engine. The queries that are submitted by our application should be optimized.
- How do we optimize queries? Optimizing queries is not trivial. This requires an understanding of SQL and the database engine that it targets.
- SQL engines have their profiling tools. You can determine execution time of an SQL query using a database profiling tool.
- Each database engine has its own profiling tools. Also each database engine has different profiling capabilities.

Retrieving SQL Queries from SQLAlchemy

- If you are using SQLAlchemy then you are not creating SQL queries but creating Python code. It is SQLAlchemy that generates an SQL query and submits it to the database. Hence you may want to print the SQL query to the console and then you can execute it against a profiler.
- The following line prints an SQL query for a Postgresql database.

```
print ins.compile(dialect=postgresql.dialect())
```

- Note: SQL queries are dependent upon the database engine that it executes against. Hence, it should be compiled for that specific database engine using a “dialect” parameter.
- The code below is the same as in Module 6 with only one line added. **Can you guess which line has been added and what it does?**

```
from sqlalchemy import create_engine, MetaData, Table, Column, Integer, String
engine = create_engine('sqlite:/// ccps530.db', echo = True)
```

```
meta = MetaData()

users = Table(
   'users', meta,
   Column('id', Integer, primary_key = True),
   Column('name', String),
   Column('email', String),
)

ins = users.insert()
ins = users.insert().values(name = 'Mike Jones', email = 'mike.jones@global.com’)
print ins.compile(dialect=postgresql.dialect())
conn = engine.connect()
result = conn.execute(ins)
```

Werkzeug

- Werkzeug is a comprehensive WSGI web application library and it includes an interactive debugger and a built-in profiler.
- To use Werkzeug you need to install it using pip command

```
pip install werkzeug
```

- The profiler can be used by any application by simply configuring and enabling it within the application.

Enabling Werkzeug

-
  ```
  if __name__ == '__main__':
  from werkzeug.middleware.profiler import ProfilerMiddleware
  app.wsgi_app = ProfilerMiddleware(app.wsgi_app, restrictions=[5],
  profile_dir='./profile')
  app.run(debug=True)
  ```

  Above, we have the code that enables Werkzeug to be used within your application.

- The profiler has 2 parameters
  a. A number of slowest performing functions to show. We have configured the “restrictions” parameter with a value of 5.
  b. A directory where to put details of function calls. We have configured “profile_dir” with the directory “profile”.
- When the application runs the output can be seen on the console. This can be seen below.

Werkzeug Output

```
PATH: '/my_class'
          8887 function calls (8615 primitive calls) in 0.038 seconds

   Ordered by: internal time, call count
   List reduced from 994 to 5 due to restriction <5>
```

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     3    0.004     0.001     0.004     0.001 {method 'recv_into' of '_socket.socket'
objects}
     2    0.001     0.000     0.001     0.000 /Users/as/.virtualenvs/flask-
perf/lib/python3.6/site-packages/sqlalchemy/orm/interfaces.py:558(create_row_processor)
      28    0.001     0.000     0.001     0.000 {method 'update' of 'dict' objects}
      57    0.001     0.000     0.001     0.000
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/_weakrefset.py:58(__iter__)
      309   0.001     0.000     0.001     0.000 {built-in method builtins.getattr}
```

Profiling Files (*.prof)

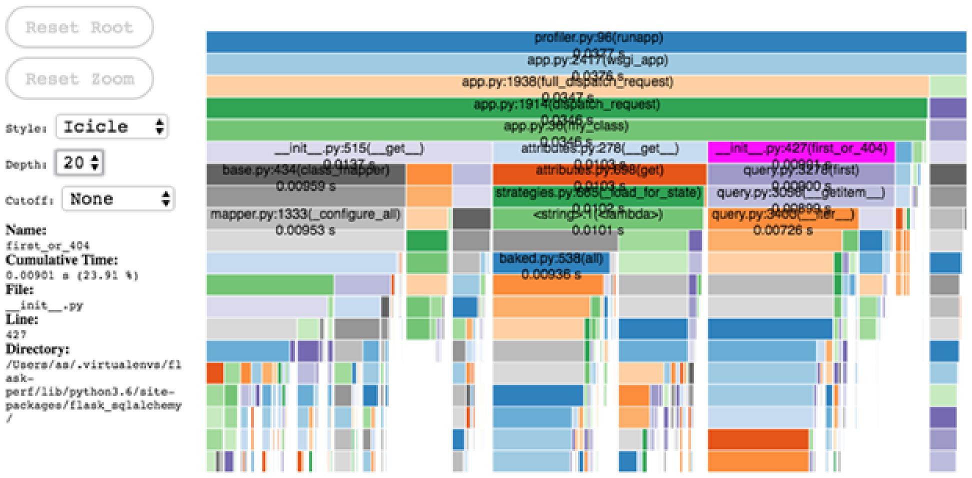
- Profiling files (*.prof) generated by Werkzeug may be difficult to read but can be easily viewed with a graphical viewer.
- A graphical viewer is snakeviz. Snakeviz can be installed with pip.

```
pip install snakeviz
```

- Snakeviz can be invoked on a single .prof file. These files are in the profile directory.

```
snakeviz profile/GET.my_class.38ms.1588862067.prof
```

- In Figure 12.1 below, you can see the output of snakeviz from the above profiling file.



Style: Icicle
Depth: 20
Cutoff: None
Name:
first_or_404
Cumulative Time:
0.00901 s (23.91 %)
File:
__init__.py
Line:
427
Directory:
/Users/as/.virtualenvs/flask-perf/lib/python3.6/site-packages/flask_sqlalchemy/

Figure 12.1. Snakeviz profiling output from accessing a route.

[Source: Alexey Smirnov - How to increase Flask performance]

- An alternative to snakeviz is gprof2dot.
- Gprof2dot can be installed using pip.

```
pip install gprof2dot
```

- Gprof2dot converts a .prof file to a dot file and xdot draws a graph from the dot file.
- The following command shows how use gprof2dot, generate a dot file and invoke xdot.

```
gprof2dot -f pstats profile/GET.my_class.38ms.1588862067.prof -o calling_graph.dot
xdot calling_graph.dot
```

- Note the above command is executed on the same profile file as werkzeug. See Figure 12.2 below for the output.

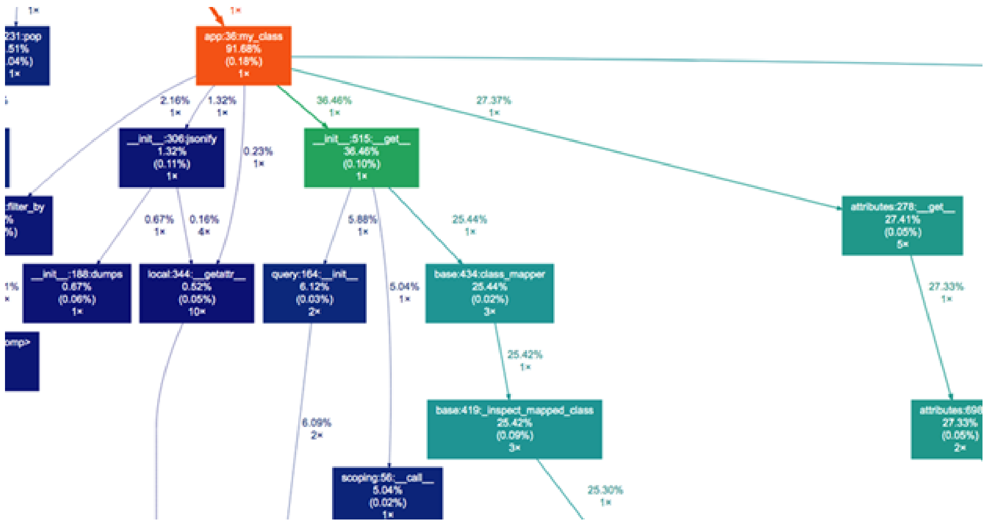


Figure 12.2. gprof2dot profiling output from accessing a route.

Source: Alexey Smirnov – How to increase Flask performance

Video

Please watch the following video, Understanding Application Performance Using Werkzeug. This video demonstrates how to profile web application routes.

Understanding Application Performance Using Werkzeug
TMU Video

Flask-Debug Toolbar

- Flask-Debug toolbar is a higher level profiler. It can be installed using pip.

```
pip install flask-debugtoolbar
```

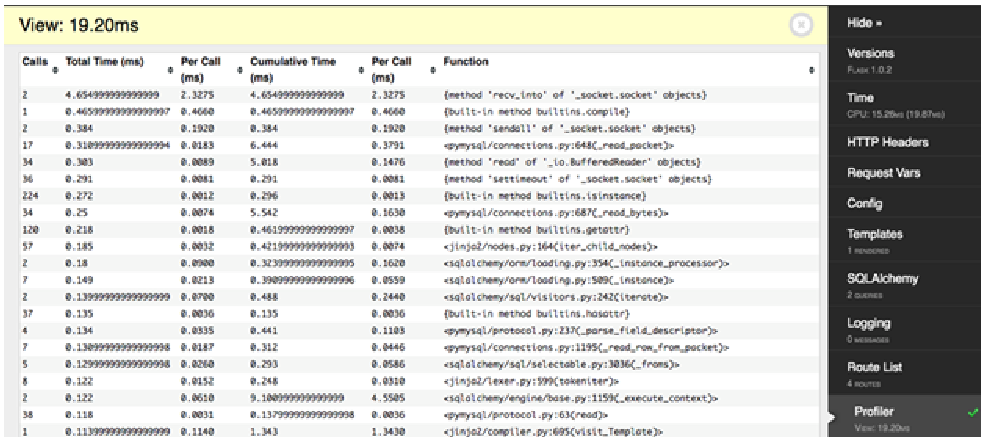
- The Fask-Debug toolbar has to be enabled with the application

```
from flask_debugtoolbar import DebugToolbarExtension

app.config['SECRET_KEY'] = 'verysecretsuchwow'
app.config['DEBUG_TB_PROFILER_ENABLED'] = True
app.debug = True

toolbar = DebugToolbarExtension(app)
```

- Every response with have a toolbar on the right-hand side of a browser. The output is in Figure 12.3 below.



View: 19.20ms

Calls	Total Time (ms)	Per Call (ms)	Cumulative Time (ms)	Per Call (ms)	Function
2	4.654999999999999	2.3275	4.654999999999999	2.3275	{method 'recv_into' of '_socket.socket' objects}
1	0.46599999999999997	0.4660	0.46599999999999997	0.4660	{built-in method builtins.compile}
2	0.384	0.1920	0.384	0.1920	{method 'sendall' of '_socket.socket' objects}
17	0.31099999999999994	0.0183	6.444	0.3791	<pymysql/connections.py:648(_read_packet)>
34	0.303	0.0089	5.018	0.1476	{method 'read' of '_io.BufferedReader' objects}
36	0.291	0.0081	0.291	0.0081	{method 'settimeout' of '_socket.socket' objects}
224	0.272	0.0012	0.296	0.0013	{built-in method builtins.isinstance}
34	0.25	0.0074	5.542	0.1630	<pymysql/connections.py:687(_read_bytes)>
120	0.218	0.0018	0.46199999999999997	0.0038	{built-in method builtins.getattr}
57	0.185	0.0032	0.42199999999999993	0.0074	<jinja2/nodes.py:164(iter_child_nodes)>
2	0.18	0.0900	0.32399999999999995	0.1620	<sqlalchemy/orm/loading.py:354(_instance_processor)>
7	0.149	0.0213	0.39099999999999996	0.0559	<sqlalchemy/orm/loading.py:509(_instance)>
2	0.13999999999999999	0.0700	0.488	0.2440	<sqlalchemy/sql/visitors.py:242(iterate)>
37	0.135	0.0036	0.135	0.0036	{built-in method builtins.hasattr}
4	0.134	0.0335	0.441	0.1103	<pymysql/protocol.py:237(_parse_field_descriptor)>
7	0.13099999999999998	0.0187	0.312	0.0446	<pymysql/connections.py:1195(_read_row_from_packet)>
5	0.12999999999999998	0.0260	0.293	0.0586	<sqlalchemy/sql/selectable.py:3036(_froms)>
8	0.122	0.0152	0.248	0.0310	<jinja2/lexer.py:599(tokeniter)>
2	0.122	0.0610	9.100999999999999	4.5505	<sqlalchemy/engine/base.py:1159(_execute_context)>
38	0.118	0.0031	0.13799999999999998	0.0036	<pymysql/protocol.py:63(read)>
1	0.11399999999999999	0.1140	1.343	1.3430	<jinja2/compiler.py:695(visit_Template)>

Figure 12.3. Flask-Debug toolbar profiling output from accessing a route.

[Source: Alexey Smirnov – How to increase Flask performance]

Long Description +

The toolbar allows you to see the stack trace of the function call triggered by a route or url web browser access. The stack trace displays the time each function takes to execute.

API Monitoring

- To monitor higher level calls such as invocation of URL endpoints there are two plugins names Flask-MonitoringDashboard and flask-profiler.
    - Note: Flask-MonitoringDashboard works on top of flask-profiler.

```
pip install flask_profiler Flask-MonitoringDashboard
```

- To enable the Flask-MonitoringDashboard add the following code to your application.

```
import flask_monitoringdashboard as dashboard
dashboard.bind(app)
```

- The dashboard is available under/dashboard (default credentials are admin/admin). The output is in Figure 12.4 below. In the API performance section, you can find statistics showing response time distributions for your endpoints.

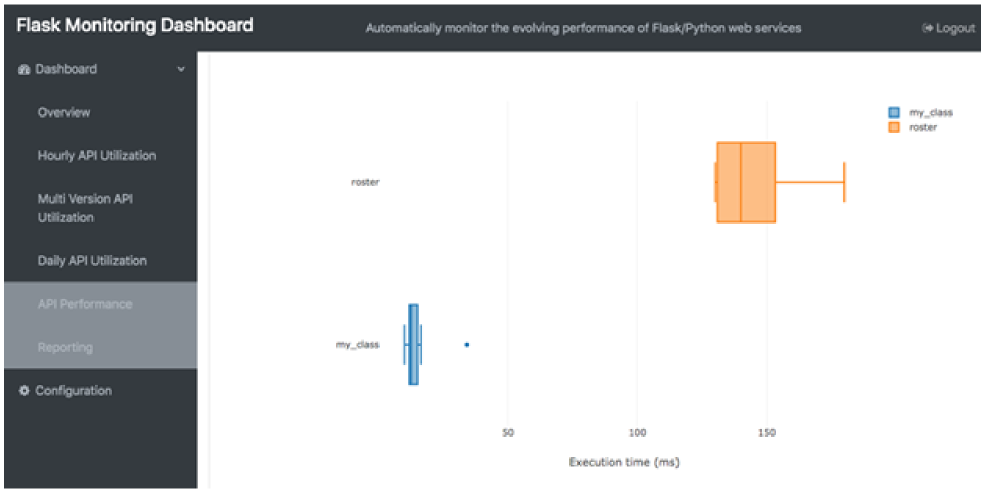


Figure 12.4. Flask Monitoring Dashboard profiling output from accessing different routes of your web application.

[Source: Alexey Smirnov – How to increase Flask performance]()

Video

Please watch the following video, Understanding Application Performance Using Flask Monitoring Dashboard. This video demonstrates how to profile web application routes.

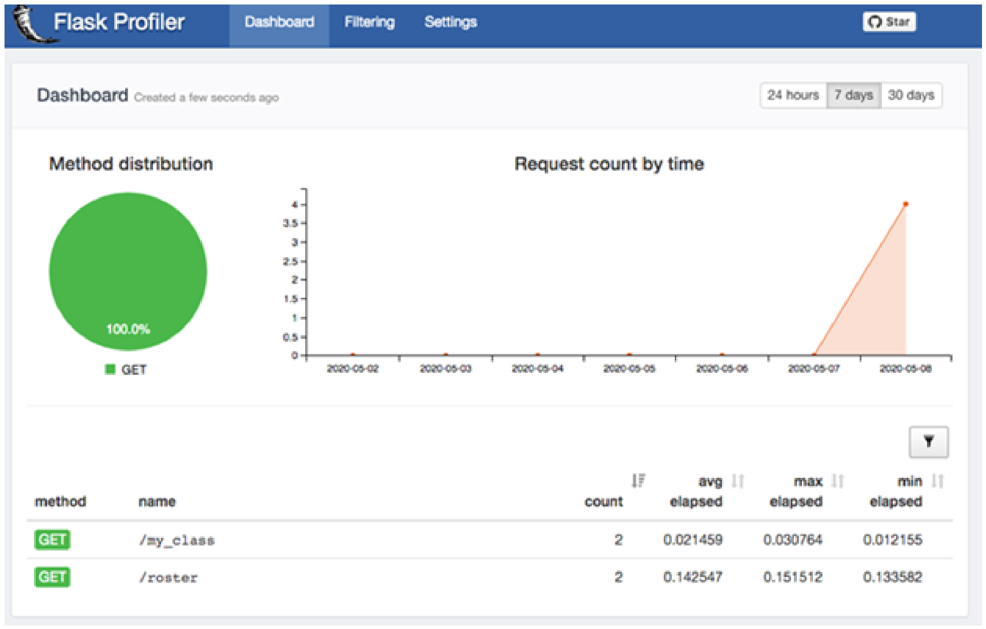
Understanding Application Performance Using Flask Monitoring Dashboard
TMU Video

Flask Profiler

The flask profiler is harder to use for beginners due to its many configuration options. Beginners may find the number of configuration options intimidating. Below you will find code that enables the flask profiler in your application. Note the number of options available for simple usage. The profiler is available under /flask-profile with default credential admin/admin.

```
from flask_profiler import Profiler
app.config["flask_profiler"] = {
     "enabled": app.config["DEBUG"],
     "storage": {
    "engine": "sqlite"
     },
     "basicAuth":{
    "enabled": True,
    "username": "admin",
    "password": "admin"
     },
     "ignore": [
    "^/static/.*"
     ]
}
Profiler(app)
```

In Figure 12. 5 below you can see profile information for two routes /my_class and /roster.



method	name	count	avg elapsed	max elapsed	min elapsed
GET	/my_class	2	0.021459	0.030764	0.012155
GET	/roster	2	0.142547	0.151512	0.133582

Figure 12.5. Flask Profiler profiling output from accessing different routes of your web application. This is the same information as Figure 12.4, but in a different format.

[Source: Alexey Smirnov – How to increase Flask performance]

Summary

Profiling allows you to see the execution time of your application. In our case, we are interested in the execution time of the routes made available by our web application. Your application is dependent on implementing business logic that accesses datastores and third party services. It is difficult to determine the execution efficiency of your application without profiling tools. In this module we have seen different profiling tools. It is important to notice that there is overlap in the feature availability of the different profiling tools. Hence, it is up to you to determine which profiling tool(s) you will use. One simple but important criteria is which profiling tools are easy to work and what data you need from the profiling tools.

Assessments

Each week, you'll find here reminders of assignments or labs that you should be working on. For Week 12:

- Lab 12: Optimizing your application (5% of the final grade) is due in 7 days, i.e., the end of day Friday of Week 12.

Reminder

You have a weekly Zoom session. The date/timing for your Zoom session can be found in the Course Schedule in the Course Outline. The link to the Zoom session will be provided separately to you by your instructor. The Zoom session will consist of 20–30 minutes of lecture, followed by questions and answers (Q&A), followed by a lab period. You may ask questions about the previous or current week's content or labs.

Please bring any questions to the weekly Zoom session. However, if you have any questions during the week outside of the Zoom session, you may post them in the Discussion Board. Your instructor may respond in the Discussion board or during the weekly Zoom session.

Click-n-reveal: **Where is the Discussion board?**

Click on the "Communication" area at the top main course menu, and select Discussions from the dropdown menu.

References

None for this week.