Load Testing & Bottlenecks

Teacher Version

Software Architecture

April 29, 2025

Brae Webb & Richard Thomas

Aside

Github Classroom links for this practical can be found on Edstem `https://edstem.org/au/courses/21491/discussion/2429006`

Warning

This practical **cannot** be completed on Windows. The Docker provider for Terraform encounters an error when creating an image on Windows. You will need to use WSL instead of the native Windows environment.
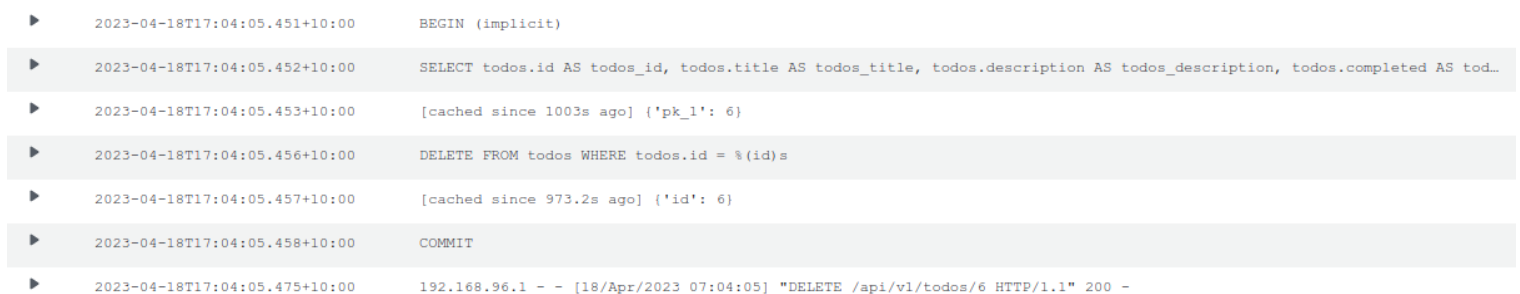
1 This Week

Our goal is to:

- Introduce structured logging to TaskOverflow.

- Re-deploy TaskOverflow to AWS.

- Write k6 tests to ensure TaskOverflow can handle given scenarios.

- Uncover and fix bottlenecks in the TaskOverflow application using the logs.

2 Watchtower

In this course we have repeatedly claimed that there is value in monitoring and logging. This week, we aim to prove it by using logging to help diagnose issues with a deployment of TaskOverflow. For this task, we have integrated Watchtower[1] into the project. Watchtower is a Python library that allows us to send logs to AWS CloudWatch, allowing us to monitor an application's performance in real time.

Currently the project is configured to log API accesses made with Flask and to log database queries created by SQL Alchemy. This results in an unstructured log stream as seen in Figure 1.

```
▶   2023-04-18T17:04:05.451+10:00      BEGIN (implicit)

▶   2023-04-18T17:04:05.452+10:00      SELECT todos.id AS todos_id, todos.title AS todos_title, todos.description AS todos_description, todos.completed AS tod...

▶   2023-04-18T17:04:05.453+10:00      [cached since 1003s ago] {'pk_1': 6}

▶   2023-04-18T17:04:05.456+10:00      DELETE FROM todos WHERE todos.id = %(id)s

▶   2023-04-18T17:04:05.457+10:00      [cached since 973.2s ago] {'id': 6}

▶   2023-04-18T17:04:05.458+10:00      COMMIT

▶   2023-04-18T17:04:05.475+10:00      192.168.96.1 - - [18/Apr/2023 07:04:05] "DELETE /api/v1/todos/6 HTTP/1.1" 200 -
```

Figure 1: An example of logs made to AWS CloudWatch for a DELETE request in the TaskOverflow API.

Getting Started

1. Using the GitHub Classroom link for this practical, provided on edstem, create a repository to work within.

2. Start your Learner Lab and copy the AWS Learner Lab credentials into a credentials file in the root of the repository.

What's New   We are returning to TaskOverflow roughly from the state at the end of the *Scaling Stateless Components* practical [1]. The following notable changes have been made:

- `Watchtower` has been installed as a dependency.

- In `docker-compose.yml`, we mount the `credentials` file to `/root/.aws/credentials`. This allows local testing of `watchtower` to log to AWS CloudWatch.

- Logging has been introduced for Flask and SQL Alchemy.

---

[1] https://pypi.org/project/watchtower/

Our first task will be to replicate the above logs in Figure 1. Once you have copied `credentials` into the project root directory, start docker compose with:

```
$ docker compose up
```

```
app-1 | * Serving Flask app 'todo'
app-1 | * Debug mode: on
app-1 | INFO:werkzeug:WARNING: This is a development server. Do not use it in a
    production deployment. Use a production WSGI server instead.
app-1 | * Running on all addresses (0.0.0.0)
app-1 | * Running on http://127.0.0.1:6400
app-1 | * Running on http://172.18.0.3:6400
app-1 | INFO:werkzeug:Press CTRL+C to quit
app-1 | INFO:werkzeug: * Restarting with stat
app-1 | INFO:botocore.credentials:Found credentials in shared credentials file: ~/.
    aws/credentials
app-1 | INFO:sqlalchemy.engine.Engine:select pg_catalog.version()
app-1 | INFO:sqlalchemy.engine.Engine:[raw sql] {}
app-1 | INFO:sqlalchemy.engine.Engine:select current_schema()
app-1 | INFO:sqlalchemy.engine.Engine:[raw sql] {}
app-1 | INFO:sqlalchemy.engine.Engine:show standard_conforming_strings
app-1 | INFO:sqlalchemy.engine.Engine:[raw sql] {}
app-1 | INFO:sqlalchemy.engine.Engine:BEGIN (implicit)
app-1 | INFO:sqlalchemy.engine.Engine:SELECT pg_catalog.pg_class.relname
app-1 | FROM pg_catalog.pg_class JOIN pg_catalog.pg_namespace ON pg_catalog.
    pg_namespace.oid = pg_catalog.pg_class.relnamespace
app-1 | WHERE pg_catalog.pg_class.relname = %(table_name)s::VARCHAR AND pg_catalog.
    pg_class.relkind = ANY (ARRAY[%(param_1)s::VARCHAR, %(param_2)s::VARCHAR, %(
    param_3)s::VARCHAR, %(param_4)s::VARCHAR, %(param_5)s::VARCHAR]) AND pg_catalog.
    pg_table_is_visible(pg_catalog.pg_class.oid) AND pg_catalog.pg_namespace.nspname
    != %(nspname_1)s::VARCHAR
app-1 | INFO:sqlalchemy.engine.Engine:[generated in 0.00015s] {'table_name': 'todos',
     'param_1': 'r', 'param_2': 'p', 'param_3': 'f', 'param_4': 'v', 'param_5': 'm',
    'nspname_1': 'pg_catalog'}
app-1 | INFO:sqlalchemy.engine.Engine:COMMIT
app-1 | WARNING:werkzeug: * Debugger is active!
app-1 | INFO:werkzeug: * Debugger PIN: 123-456-789
```

You should see logs similar to the above. Notice that information about Flask is prefixed with `INFO:werkzeug` and information about SQL Alchemy is prefixed with `INFO:sqlalchemy.engine.Engine`. These prefixes indicate into which stream the logs are put.

Open the AWS CloudWatch console and go to `Logs > Log groups` on the side panel. You should see a log group called `taskoverflow`. If you click on that group you will see log streams associated with `INFO:werkzeug` and `INFO:sqlalchemy.engine.Engine`.

2.1 Structured Logging

Our first task will be to convert the current logging into a structured logging format. As we saw in logging lecture [2], structured logging can be as simple as logging a JSON object. This allows logging services to quickly filter through logs based on criteria selected from the object's fields.

In `todo/__init__.py` we have the following code within the `create_app` function. This code configures watchtower to log to AWS CloudWatch.

```
» cat todo/__init__.py
def create_app(config_overrides=None):
    ...
    handler = watchtower.CloudWatchLogHandler(
        log_group_name="taskoverflow",
        boto3_client=boto3.client("logs", region_name="us-east-1")
    )
    app.logger.addHandler(handler)
    ...
```

We want to wrap all our logs in JSON objects and inject metadata into these objects. This makes it easier to search the logs. To do this, we will create a custom log formatter that is a subclass of `watchtower.CloudWatchLogFormatter`.

```
» cat todo/__init__.py
from flask import has_request_context, request
...

class StructuredFormatter(watchtower.CloudWatchLogFormatter):
    def format(self, record):
        record.msg = {
```

```
            'timestamp': record.created,
            'location': record.name,
            'message': record.msg,
        }
        if has_request_context():
            record.msg['request_id'] = request.environ.get('REQUEST_ID')
            record.msg['url'] = request.environ.get('PATH_INFO')
            record.msg['method'] = request.environ.get('REQUEST_METHOD')
        return super().format(record)

def create_app(config_overrides=None):
    ...
    handler = ...
    handler.setFormatter(StructuredFormatter())
    ...
```

These are just some metadata fields we might want to add to our logging. We may also consider `socket.gethostname()` to identify the instance handling a request, or other helpful information.

2.2   Correlation IDs

Correlation IDs are a mechanism to help understand the path of a request, event, message, etc. through a system. When logging it is often helpful to be able to trace the execution of a particular request.

For our system we will generate a new random identifier for each incoming request. This identifier will be included as part of the logging metadata. For a system with multiple API endpoints requests will often come with an established REQUEST_ID header.

```
» cat todo/__init__.py
import uuid
...

def create_app(config_overrides=None):
    ...
    requests = logging.getLogger("requests")
    requests.addHandler(handler)

    @app.before_request
    def before_request():
        request.environ['REQUEST_ID'] = str(uuid.uuid4())
        requests.info("Request started")

    @app.after_request
    def after_request(response):
        requests.info("Request finished")
        return response
    ...
```

The code above in `create_app` will generate a unique identifier for each incoming request. It will also log when a request starts being processed and when it finishes processing.

2.3 Verify and Refine

Ensure that the above modifications are working as expected by returning to the AWS CloudWatch logs. Launch the service locally and make some API requests.

You should now be able to go to the `Log Insights` interface from the sidebar. This interface allows you to query structured logging data. Try the following query to ensure that your logs seem sensible.

```
fields request_id, url, method, @timestamp, message.message, @message, @method
| filter not isempty(request_id)
| limit 400
```

Take the time to refine your logging now to produce logs that are easy to search through and understand the trace of your API requests.

3 Load Testing

We will now generate some load on our API using k6 and see how it performs. We briefly saw k6 at the end of the *Scaling Stateless Components* practical [1]. To review, k6 is a load testing tool that is written in Go but provides an interface using a subset of JavaScript.

It will help us to generate a large number of concurrent API requests. If you have not already done so, install k6 from: https://k6.io/docs/get-started/installation/

3.1 Scenario

We want to simulate the following scenario using our testing framework.

Indecisive planners and studious reviewers   Our TaskOverflow application is being used at a point in semester where most students have already setup their tasks. These students are routinely visiting the website and listing the tasks they have yet to complete. At the same time, there are 40 very indecisive students who, quite late in semester, are trying to setup their tasks for the rest of semester. They continually create a task, realise that they have mis-typed, delete the task, and start over.

As the tasks are shared globally amongst all students, the indecisive planners are altering the list of task seen by the organised students.

3.2 Setup

To get started, we will create a file called `planners-and-studiers.js`. Most k6 tests start with the following imports.

```
 » cat planners-and-studiers.js

import http from "k6/http";
import { check, sleep } from "k6";

const ENDPOINT = __ENV.ENDPOINT;
```

- `http` holds the methods used to make HTTP requests,

- `check` allows us to assert the state of HTTP responses, and

- `sleep` gives us the ability to put a simulated user to sleep rather than continuously spamming the service with requests.

The ENDPOINT line retrieves the endpoint URL from the ENDPOINT environment variable.

3.3   User Simulation

The behaviour of our users will be defined by an exported JavaScript function. Our studying student will be listing out all of the tasks they have left to complete by using the `/api/v1/todos` endpoint.

```
» cat planners-and-studiers.js

export function studyingStudent() {
    let url = ENDPOINT + '/api/v1/todos';

    // What tasks do I have left to work on?
    let request = http.get(url);

    check(request, {
        'is status 200': (r) => r.status === 200,
    });

    // Alright I'll go work on my next task for around 2 minutes
    sleep(120);
}
```

Of course, this test is very basic as it only ensures the response code is 200. There is no guarantee that the returned data is sensible.

> **Info**
>
> If you would like a challenge, you can use the `randomIntBetween` function to have the studier occasionally tick off tasks.
> https://k6.io/docs/javascript-api/jslib/utils/randomintbetween/

Next we need to simulate our indecisive students. For them, we will need to make a POST request to the `/api/v1/todos` endpoint to create a task. Then we will need to use the ID given by the POST response to DELETE the mis-typed task.

```
» cat planners-and-studiers.js

export function indecisivePlanner() {
    let url = ENDPOINT + '/api/v1/todos';

    // I need to work on the CSSE6400 Cloud Assignment!
    const payload = JSON.stringify({
        "title": "CSSE6400 Clout Assignment",
        "completed": false,
        "description": "",
        "deadline_at": "2025-09-05T15:00:00",
    });
```

```
    const params = {
        headers: {
            'Content-Type': 'application/json',
        },
    };

    let request = http.post(url, payload, params);
    check(request, {
        'is status 200': (r) => r.status === 200,
    });

    sleep(10);

    // Oh no! Not the Clout assignment, the Cloud assignment!
    const wrongId = request.id;

    request = http.del('${url}/${wrongId}');

    check(request, {
        'is status 200': (r) => r.status === 200,
    });

    // I'll come back to it later :(
    sleep(10);
}
```

3.4 Configure Behaviour

We have now outlined how our user agents will interact with our API. We can use the `options` to configure how these interactions occur over time. To prevent overloading the API immediately, and allow any auto-scaling behaviour to be triggered, we will slowly increase the amount of studiers. We will have a consistent amount of planners, 20, who will perform a total of 400 corrections over the test.

```
  » cat planners-and-studiers.js

export const options = {
    scenarios: {
        studier: {
            exec: 'studyingStudent',
            executor: "ramping-vus",
            stages: [
                { duration: "1m", target: 1500 },
                { duration: "3m", target: 7500 },
                { duration: "2m", target: 0 },
            ],
        },
        planner: {
            exec: 'indecisivePlanner',
```

```
            executor: "shared-iterations",
            vus: 20,
            iterations: 400,
        },
    },
};
```

3.5   Running the Tests

Now we have a completed load test. Deploy the service to AWS:

```
$ terraform init
```

```
$ terraform apply
```

Once the deployment is finished, you should be given a URL where the endpoint is deployed. Use that endpoint to set the `ENDPOINT` environment variable and run the tests.

```
> export ENDPOINT=...
> k6 run planners-and-studiers.js
```

These tests should take about 10 minutes to run. You should see that most of the tests pass but that there will be a few failures throughout, as seen below.

```
WARN[0152] Request Failed error="Get \"http://taskoverflow-451078826.us-east-1.elb.
    amazonaws.com/api/v1/todos\": request timeout"
WARN[0183] Request Failed error="Get \"http://taskoverflow-451078826.us-east-1.elb.
    amazonaws.com/api/v1/todos\": request timeout"
WARN[0239] Request Failed error="Get \"http://taskoverflow-451078826.us-east-1.elb.
    amazonaws.com/api/v1/todos\": request timeout"
WARN[0241] Request Failed error="Delete \"$%7Burl%7D/$%7BwrongId%7D\": unsupported
    protocol scheme \"\""
WARN[0272] Request Failed error="Get \"http://taskoverflow-451078826.us-east-1.elb.
    amazonaws.com/api/v1/todos\": request timeout"
WARN[0330] Request Failed error="Get \"http://taskoverflow-451078826.us-east-1.elb.
    amazonaws.com/api/v1/todos\": request timeout"
WARN[0339] Request Failed error="Get \"http://taskoverflow-451078826.us-east-1.elb.
    amazonaws.com/api/v1/todos\": request timeout"
WARN[0371] Request Failed error="Get \"http://taskoverflow-451078826.us-east-1.elb.
    amazonaws.com/api/v1/todos\": request timeout"
WARN[0412] Request Failed error="Get \"http://taskoverflow-451078826.us-east-1.elb.
    amazonaws.com/api/v1/todos\": request timeout"
```

```
    TOTAL RESULTS

    checks_total...................: 16498 32.852733/s
    checks_succeeded...............: 92.47% 15257 out of 16498
    checks_failed..................: 7.52% 1241 out of 16498

      is status 200
        92% - 15257 / 1241

    HTTP
    http_req_duration..............................: avg=14.05s min=0s
      med=3.82s max=59.99s p(90)=49.14s p(95)=51.71s
      { expected_response:true }...................: avg=14.32s min
        =227.64ms med=3.81s max=59.72s p(90)=49.3s p(95)=51.7s
    http_req_failed................................: 5.13% 847 out of
      16498
    http_reqs......................................: 16498 32.852733/s

    EXECUTION
    iteration_duration.............................: avg=2m0s min=20.22s
        med=2m3s max=2m38s p(90)=2m10s p(95)=2m20s
    iterations.....................................: 10422 20.753496/s
    vus............................................: 3 min=3 max=7520
    vus_max........................................: 7520 min=7520 max
      =7520

    NETWORK
    data_received..................................: 1.5 GB 3.1 MB/s
    data_sent......................................: 2.3 MB 4.5 kB/s




running (08m22.2s), 0000/7520 VUs, 10422 complete and 6329 interrupted iterations
planner [=====================================] 20 VUs 08m22.2s/10m0s 400/400 shared
    iters
studier [=====================================] 0000/7500 VUs 7m0s
```

4   Debugging

Now we can start to debug our service. We desire to not only increase our 92% pass rate to 100%, but also to increase the amount of studiers and planners.

4.1   Tips

Open TaskOverflow   If you open TaskOverflow, you should find it has been left in an unusual state after the tests. See if you can find out from the logs how this occurred.

Follow an Interaction Use the correlation IDs that we have included to see where potential bottlenecks may be occurring. You can introduce more fine-grained logs to identify where time is being spent.

> **For the teacher**
>
> Solutions are roughly:
> 1. Discover that DELETE /api/v1/todos/undefined is being requested from the logs. Change const wrongId = request.id; to const wrongId = request.json().id;
> 2. Find that more tests are failing then there are timeouts. Change expected HTTP status code for POST to 201.
> 3. Recognise that even with these fixes, the requests still timeout. Upgrade the ECS task from 1 CPU / 2GB memory to maybe 4 CPU / 8GB memory.

References

[1] E. Hughes, A. Truffet, B. Webb, and R. Thomas, "Scaling stateless components," vol. 6 of *CSSE6400 Practicals*, The University of Queensland, April 2025. https://csse6400.uqcloud.net/practicals/week06.pdf.

[2] L. Le Gassick, "Logging slides," April 2024. https://bit.ly/csse6400-logging-sem1-2024.