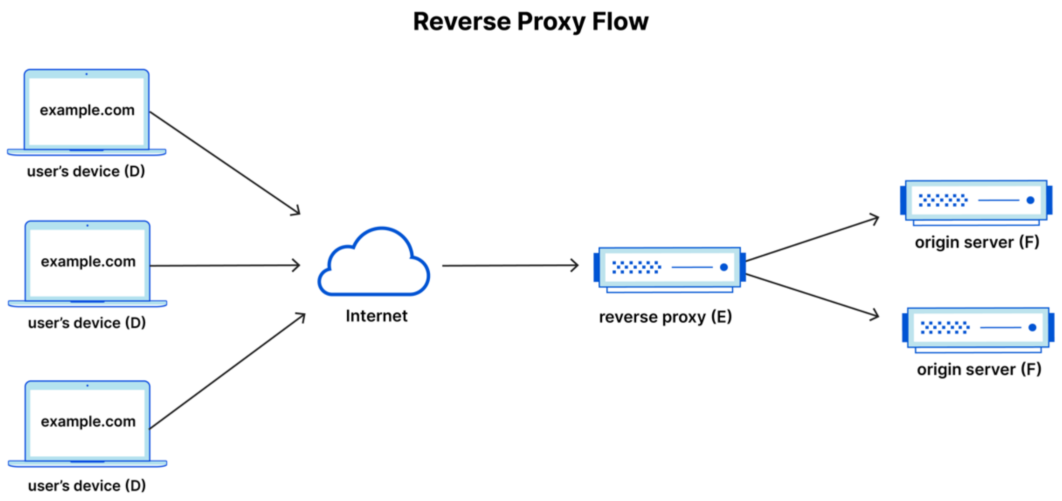
Reverse Proxy and Load Balancing with NGINX

Dr. Daniel Yellin

Reverse proxy

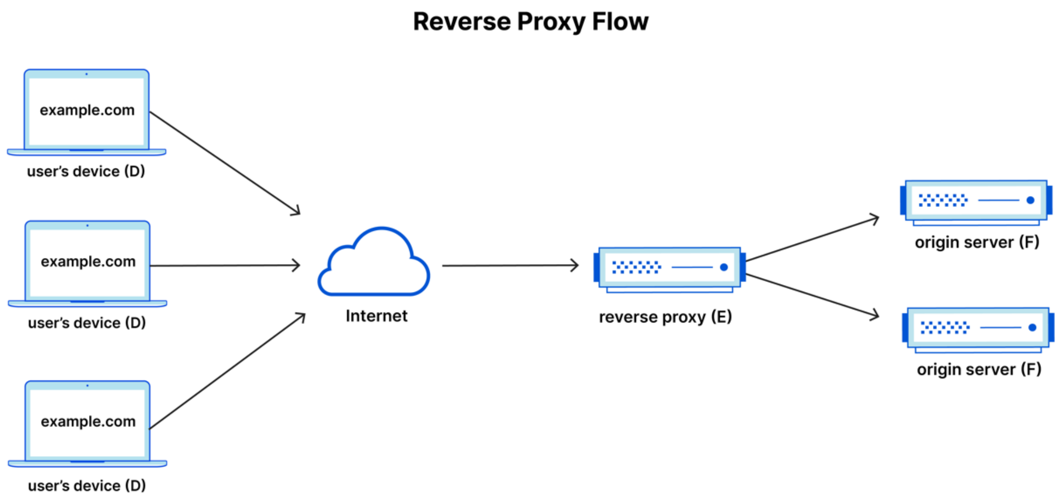
A reverse proxy sits in front of your origin servers. Clients (D) send request to the reverse proxy (E) which will then forward the request to the appropriate origin server (F).



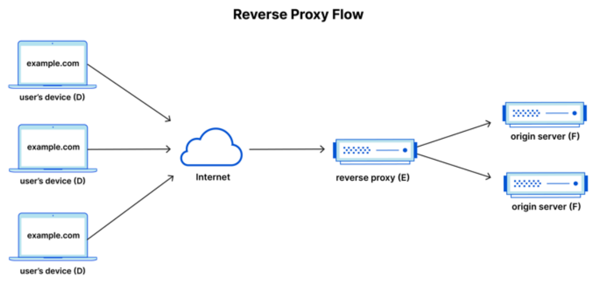
Reverse proxy

Why use a reverse proxy?

- **Protection from attacks**. As the origin svr IP address is unknown, it is harder to attack it.
- **Access control**. Only grant certain requests access to origin server.
- **Performance improvements**. The reverse proxy can perform different functions to decrease the load on the origin svr. Such as caching and decrypting/encrypting incoming/outgoing traffic.
- **Load balancing**. To distribute load across servers, and to forward to a server that is closest to the client.



NGINX as a reverse-proxy



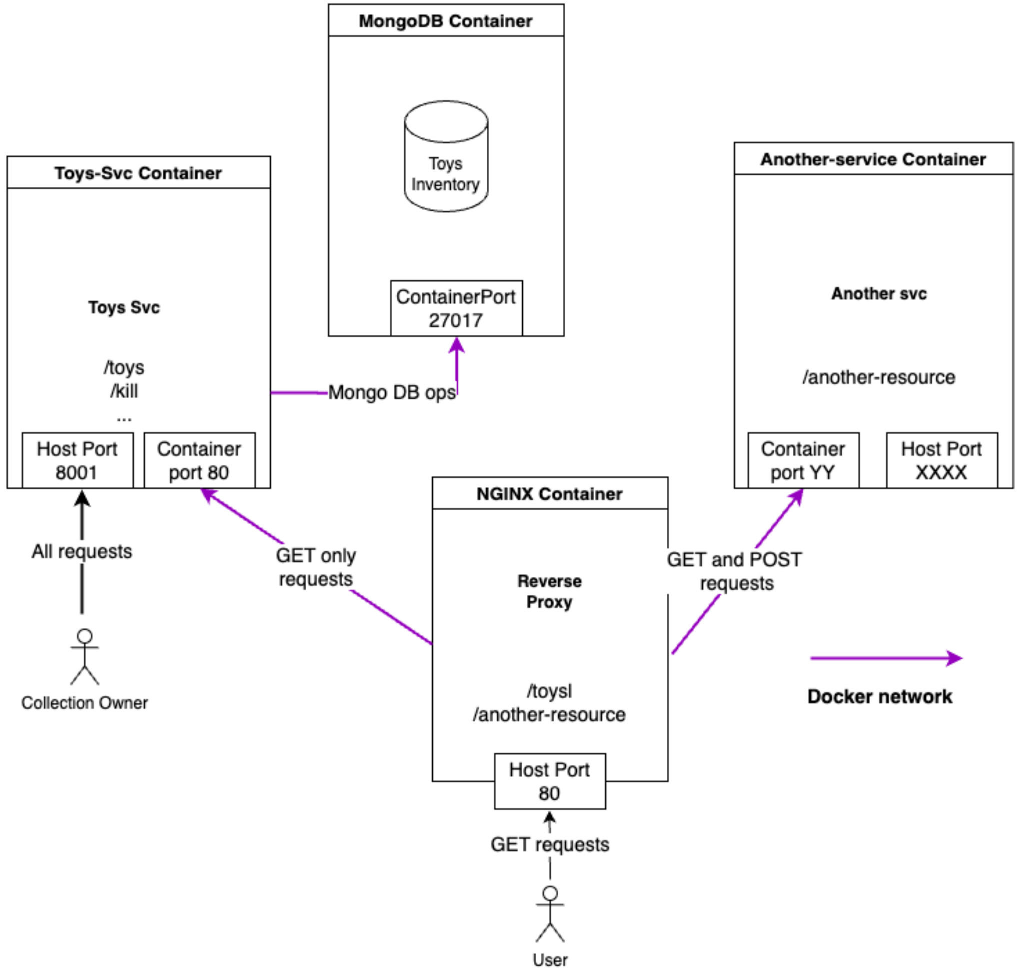
NGINX can be used as a web-server, reverse-proxy, a load balancer and more.

We use NGINX as our reverse-proxy.   We need to tell it the following things:

- What "upstream" servers are there to forward requests to and where are they located.
- Which requests should be forwarded to which upstream server
- Any rules we want stating which requests should be allowed and which should be rejected

Setting up a reverse proxy

- The diagram on the right displays our architecture.
  - There are two ways to access the toys service. One directly on port 8001. That should be on a private network only giving access to the “toys owner” who maintains the toys inventory.
  - All other users must access the toys service through the reverse-proxy running at host port 80. Such users can only issue GET requests on the toys service.
  - The NGINX reverse-proxy forwards GET requests to the toys service using the Docker network. Other types of requests such as a POST request on this resource will be refused.



Port 8001 is on a private network. For simplicity, in our implementation we have everything running on the default Docker network.

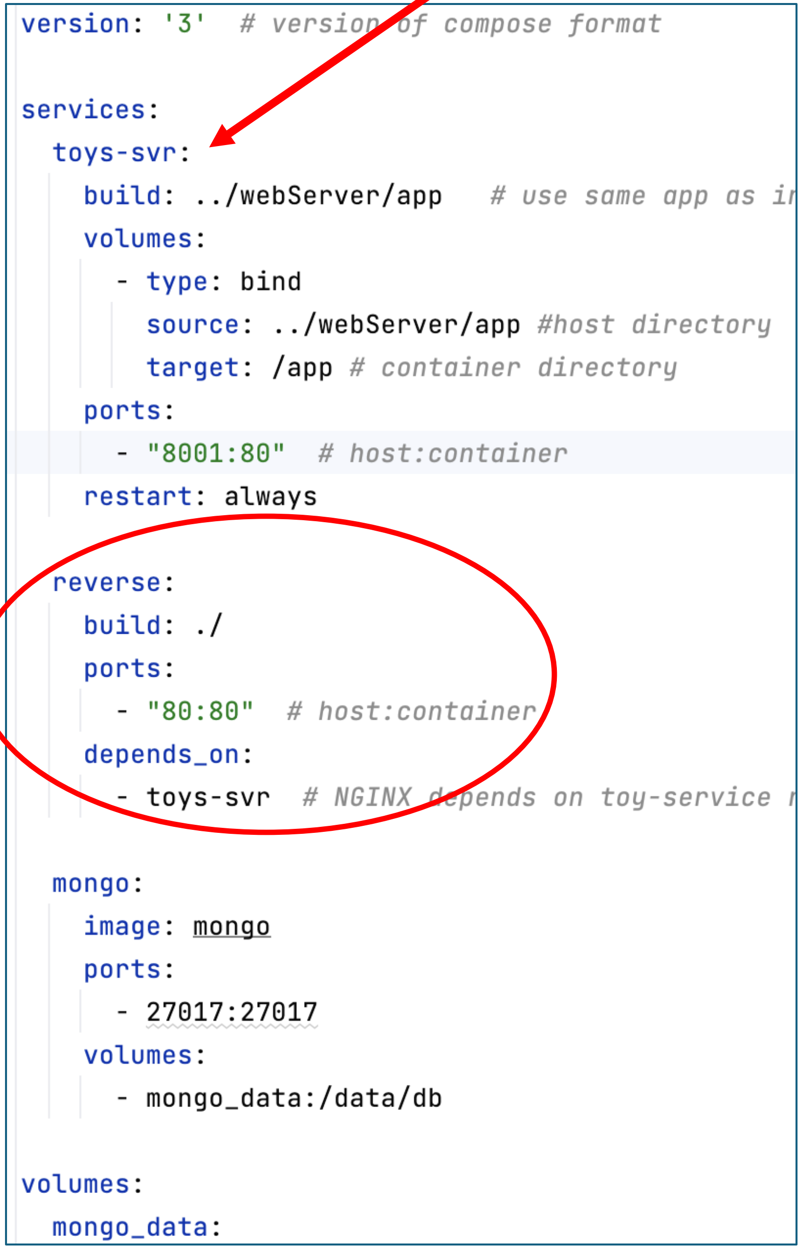
docker-compose-reverse.yml

Added additional `reverse` service to docker-compose file

- Listens on port 80 and maps to host port 80
- Depends on toys-service: cannot forward requests to toys service unless it is up and running

We need to add a configuration file so NGINX knows how to forward requests. We will do that in the NGINX Dockerfile.

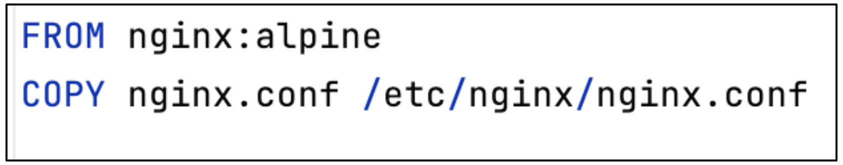
Note the name given to the toys service



```yaml
version: '3'  # version of compose format

services:
  toys-svr:
    build: ../webServer/app   # use same app as i
    volumes:
      - type: bind
        source: ../webServer/app #host directory
        target: /app # container directory
    ports:
      - "8001:80"  # host:container
    restart: always

  reverse:
    build: ./
    ports:
      - "80:80"  # host:container
    depends_on:
      - toys-svr  # NGINX depends on toy-service r

  mongo:
    image: mongo
    ports:
      - 27017:27017
    volumes:
      - mongo_data:/data/db

volumes:
  mongo_data:
```

docker-compose-

The NGINX Docker file



```
FROM nginx:alpine
COPY nginx.conf /etc/nginx/nginx.conf
```

When nginx comes up, it always looks for its configuration file in a standard directory.   Usually the config file is /etc/nginx/nginx.conf. Therefore, we need to put our nginx configuration file there so the nginx container will find it.

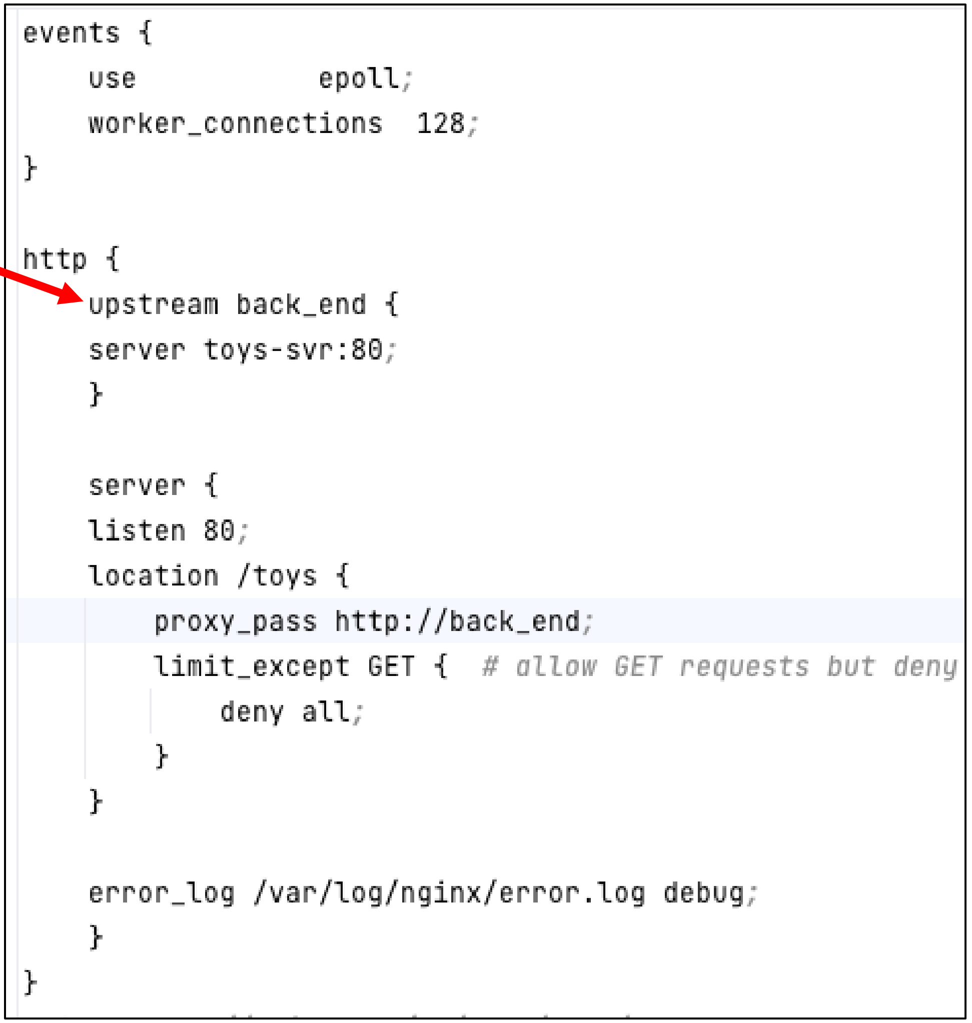
This simple nginx Dockerfile:

- tells Docker to use the nginx image (it will download it from Docker Hub if not already in your local registry), and
- copies our nginx configuration file, nginx.conf, into the image at the location nginx expects it to be at.

nginx.conf configuration file

The **upstream** directive tells NGINX which servers NGINX should forward requests to.

- In this case I list one upstream server and I give it the name.   It can be any name you want. I gave it the name **back_end**.
- If there are multiple upstream servers serving different services, put each one in its own “upstream” directive.

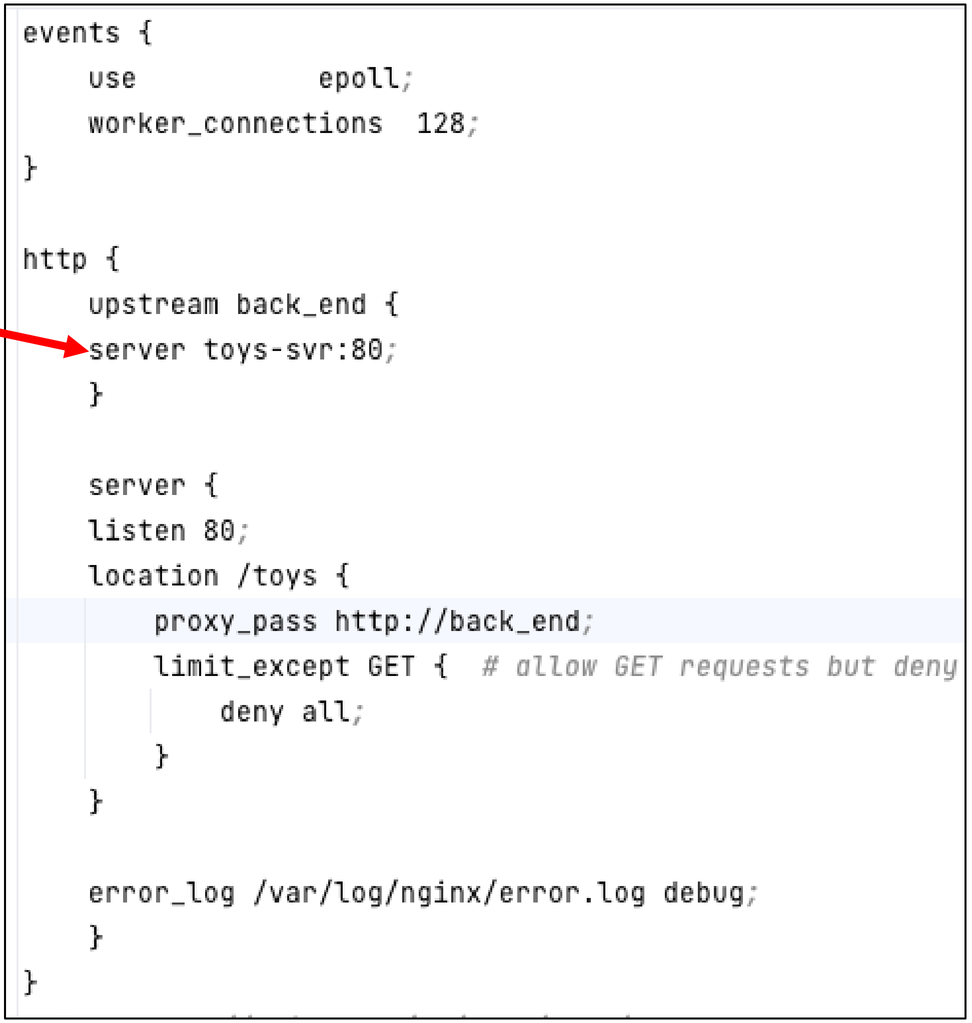


```
events {
    use           epoll;
    worker_connections  128;
}

http {
    upstream back_end {
    server toys-svr:80;
    }

    server {
    listen 80;
    location /toys {
        proxy_pass http://back_end;
        limit_except GET {  # allow GET requests but deny
            deny all;
        }
    }

    error_log /var/log/nginx/error.log debug;
    }
}
```

How to address another service

The back_end upstream server is reachable at **toys-svr:80**.

- Recall that docker-compose gave the toys service the name **toys-svr** and told it to listen at container port **80**.
- If you were instead to write “server localhost:8001” and try to reach the toys service via the localhost port, it would not work. This is because NGINX is running in its own container and its localhost is not the same as the localhost of the toys service.
- Instead NGINX contacts the toys service via the Docker network.

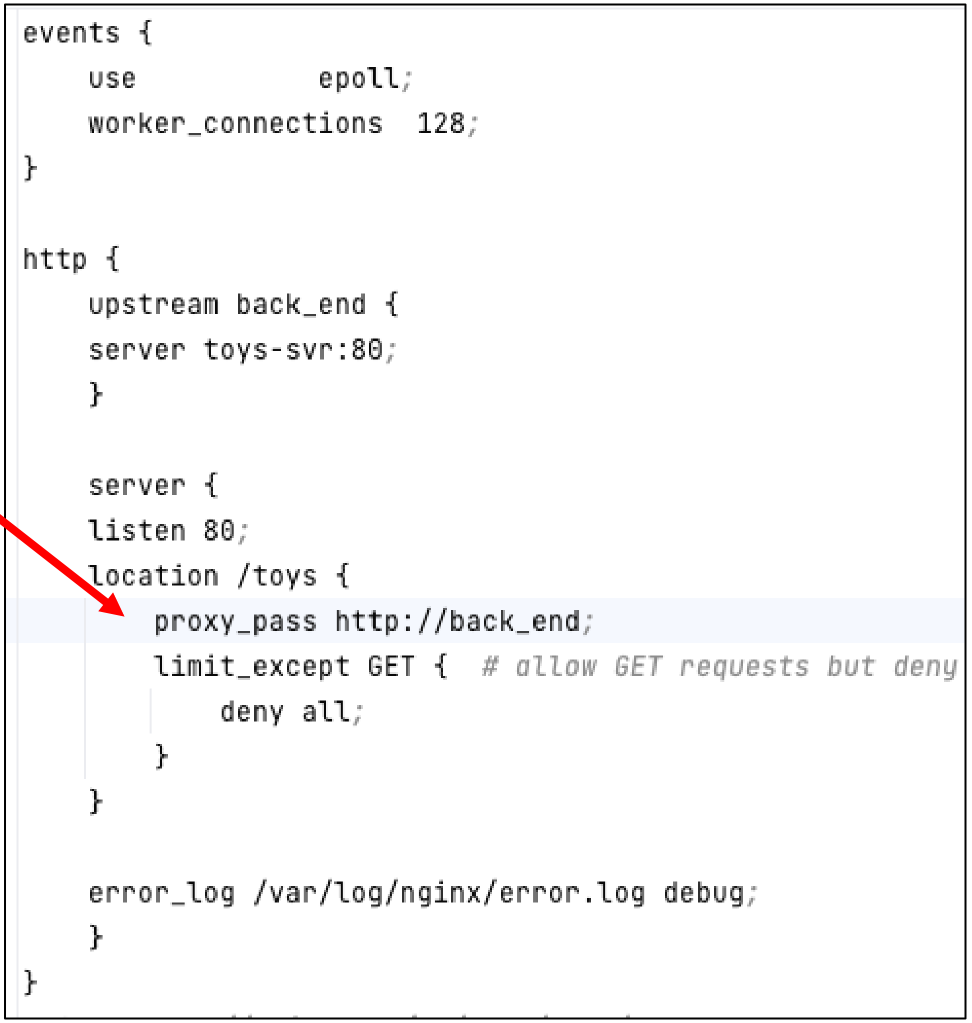


```
events {
    use              epoll;
    worker_connections  128;
}

http {
    upstream back_end {
    server toys-svr:80;
    }

    server {
    listen 80;
    location /toys {
        proxy_pass http://back_end;
        limit_except GET {  # allow GET requests but deny
            deny all;
        }
    }

    error_log /var/log/nginx/error.log debug;
    }
}
```

The location directive

The **location** directive (inside a the “server” context) tells NGINX which requests to forward to which servers.

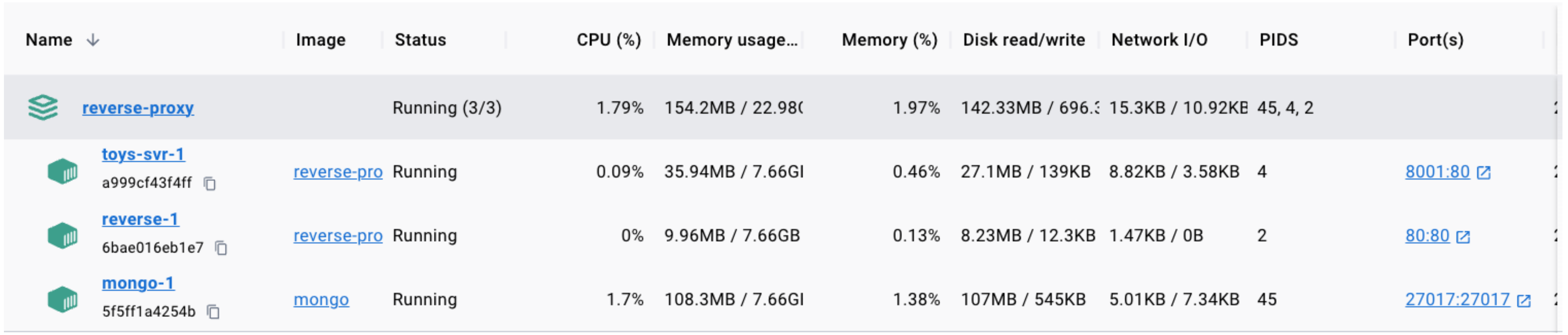
- For example, it tells NGINX that requests to the /toys resource should be forwarded to the back_end service defined above in the “upstream” context
- The **limit_except** directive tells NGINX which requests to accept (by default it will accept all). In this case, I am telling it to only forward GET requests and to deny all others.
- If you have additional resources and rules, you can have additional location blocks.



```
events {
    use             epoll;
    worker_connections  128;
}

http {
    upstream back_end {
    server toys-svr:80;
    }

    server {
    listen 80;
    location /toys {
        proxy_pass http://back_end;
        limit_except GET {  # allow GET requests but deny
            deny all;
        }
    }

    error_log /var/log/nginx/error.log debug;
    }
}
```

Let’s try it out

1. To see how this works, we will start up the application using `docker compose -f docker-compose-reverse.yml up` then check that the services are running in Docker desktop.



Name	Image	Status	CPU (%)	Memory usage...	Memory (%)	Disk read/write	Network I/O	PIDS	Port(s)
reverse-proxy		Running (3/3)	1.79%	154.2MB / 22.98(	1.97%	142.33MB / 696.:	15.3KB / 10.92KE	45, 4, 2	
toys-svr-1 a999cf43f4ff	reverse-pro	Running	0.09%	35.94MB / 7.66GI	0.46%	27.1MB / 139KB	8.82KB / 3.58KB	4	8001:80
reverse-1 6bae016eb1e7	reverse-pro	Running	0%	9.96MB / 7.66GB	0.13%	8.23MB / 12.3KB	1.47KB / 0B	2	80:80
mongo-1 5f5ff1a4254b	mongo	Running	1.7%	108.3MB / 7.66GI	1.38%	107MB / 545KB	5.01KB / 7.34KB	45	27017:27017

We see all 3 services are up and running: the mongo database, the toys-svr service, and the NGINX reverse proxy.

Let’s try it out (cont)

2. Lets POST toys to the 8001 port on the local host:

```
curl --location 'http://0.0.0.0:8001/toys' \
--header 'Content-Type: application/json' \
--data '{
   "name": "blocks",
   "descr": "12 building blocks",
   "age": 3,
   "price": 18.00
}'
{"id":"675933c67290d659c5d8a281"}
```

```
curl --location 'http://0.0.0.0:8001/toys' \
--header 'Content-Type: application/json' \
--data '{
   "name": "blocks",
   "descr": "12 building blocks",
   "age": 3,
   "price": 18.00
}'
{"id":"675933fa7290d659c5d8a282"}
```

responses

Let’s try it out (cont)

3. But when we try to POST to port 80 (NGINX), we get a different response:

```
curl --location 'http://127.0.0.1:80/toys' \
--header 'Content-Type: application/json' \
--data '{
   "name": "bubbles",
   "descr": "simple bubble blower",
   "age": 1,
   "price": 3.00
}'
<html>
<head><title>403 Forbidden</title></head>
<body>
<center><h1>403 Forbidden</h1></center>
<hr><center>nginx/1.27.3</center>
</body>
```

response
Note: you can customize the error message that NGINX returns.

That’s because we disallowed POST requests on port 80 (NGINX).

Let’s try it out (cont)

4. However, as we specified, we are allowed to perform a GET request to port 80 (NGINX):

```
curl --location 'http://127.0.0.1:80/toys'
[{"age":3,"descr":"12 building blocks", "features":[], "id":"675933c67290d659c5d8a281", "name":"blocks", "price":18.0},
{"age":3,"descr":"12 building blocks", "features":[], "id":"675933fa7290d659c5d8a282", "name":"blocks", "price":18.0}]
```

response

Nginx as load balancer

NGINX is software that can be used as a web server, reverse proxy, HTTP cache, mail proxy, and load balancer. It has been adopted by several of the busiest websites — like Adobe and WordPress — for fast request processing and response delivery.

We now show how to use NGINX as a load balancer.

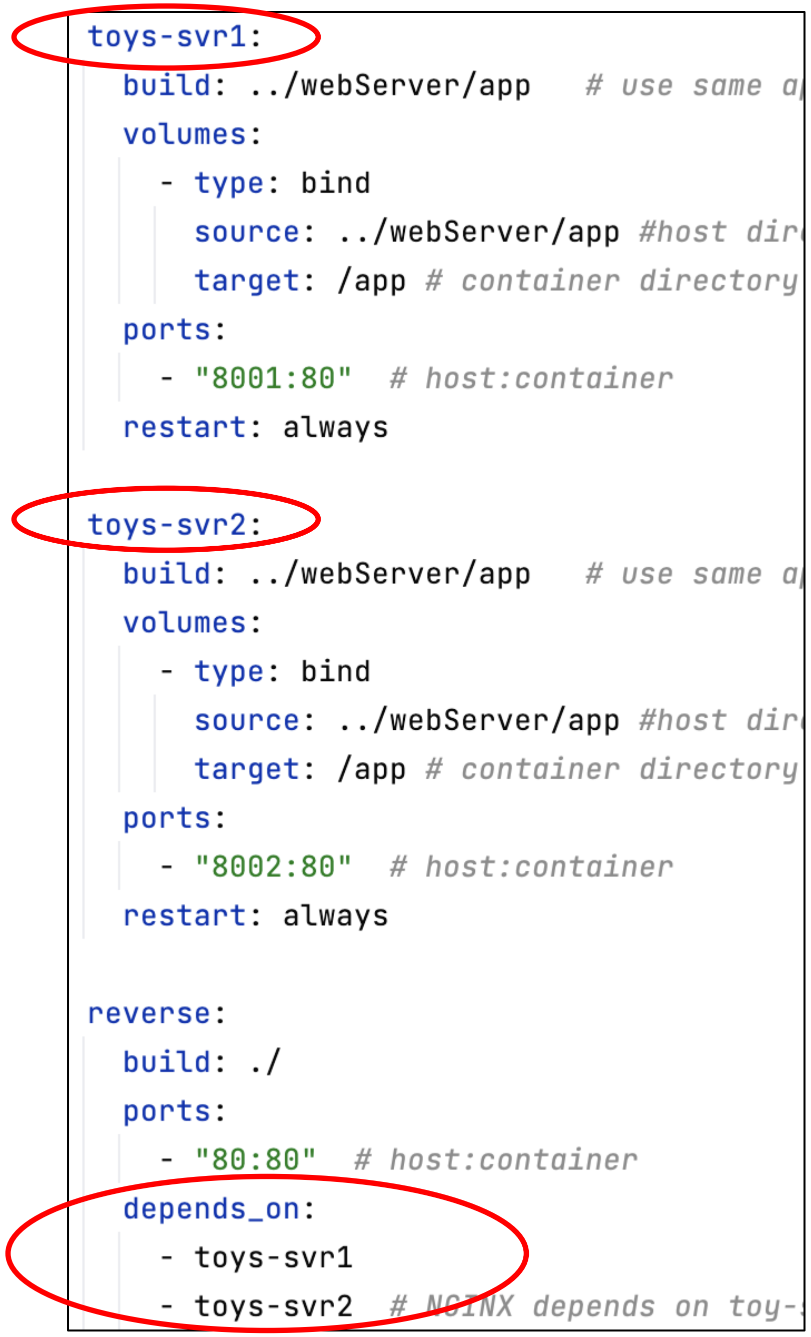
- It can be used for high-traffic websites. If properly configured, it can serve more than 10 thousand concurrent requests with low memory usage.

Add load balancing

docker-compose-lb.yml is the same as we used in the reverse proxy except:

- We create two instances of the toys service, toys-svr1 and toys-svr2. They use the same build file, but listen on different ports (8001 and 8002).
- NGINX server now depends on both of these instances.

Note: The mongo service part of the compose file is not shown

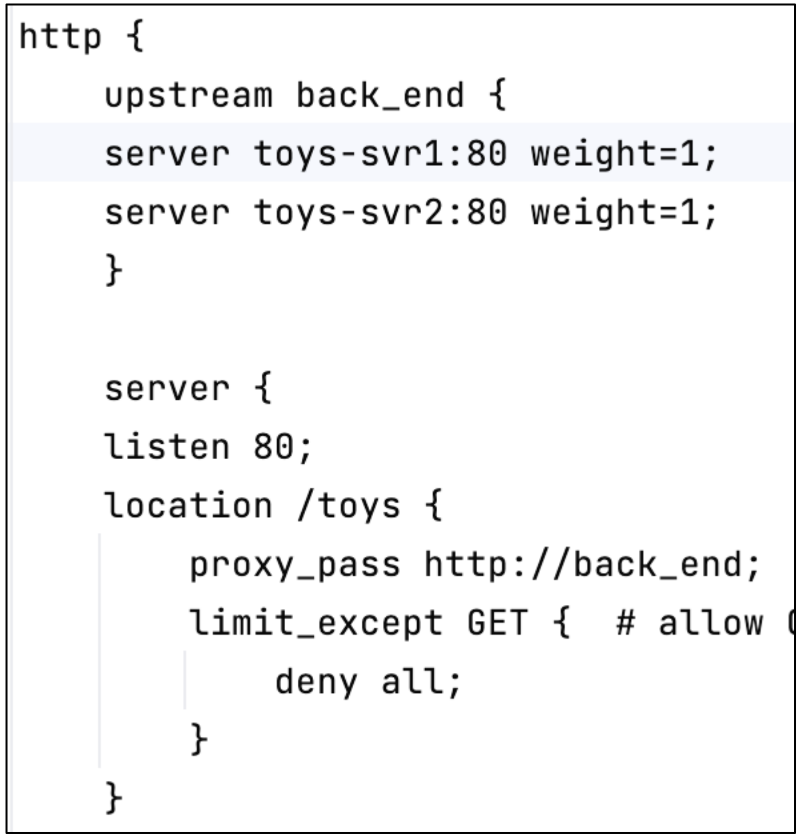


```yaml
toys-svr1:
  build: ../webServer/app    # use same a
  volumes:
    - type: bind
      source: ../webServer/app #host dir
      target: /app # container directory
  ports:
    - "8001:80"   # host:container
  restart: always

toys-svr2:
  build: ../webServer/app    # use same a
  volumes:
    - type: bind
      source: ../webServer/app #host dir
      target: /app # container directory
  ports:
    - "8002:80"   # host:container
  restart: always

reverse:
  build: ./
  ports:
    - "80:80"   # host:container
  depends_on:
    - toys-svr1
    - toys-svr2  # NGINX depends on toy-
```

Update nginx.config

nginx.config is the same as we used in the reverse proxy except:

- We create two upstream back_end servers: toys-svr1 and toys-svr2.
- We assign them weights.   In this case it says that NGINX should distribute requests to them equally.

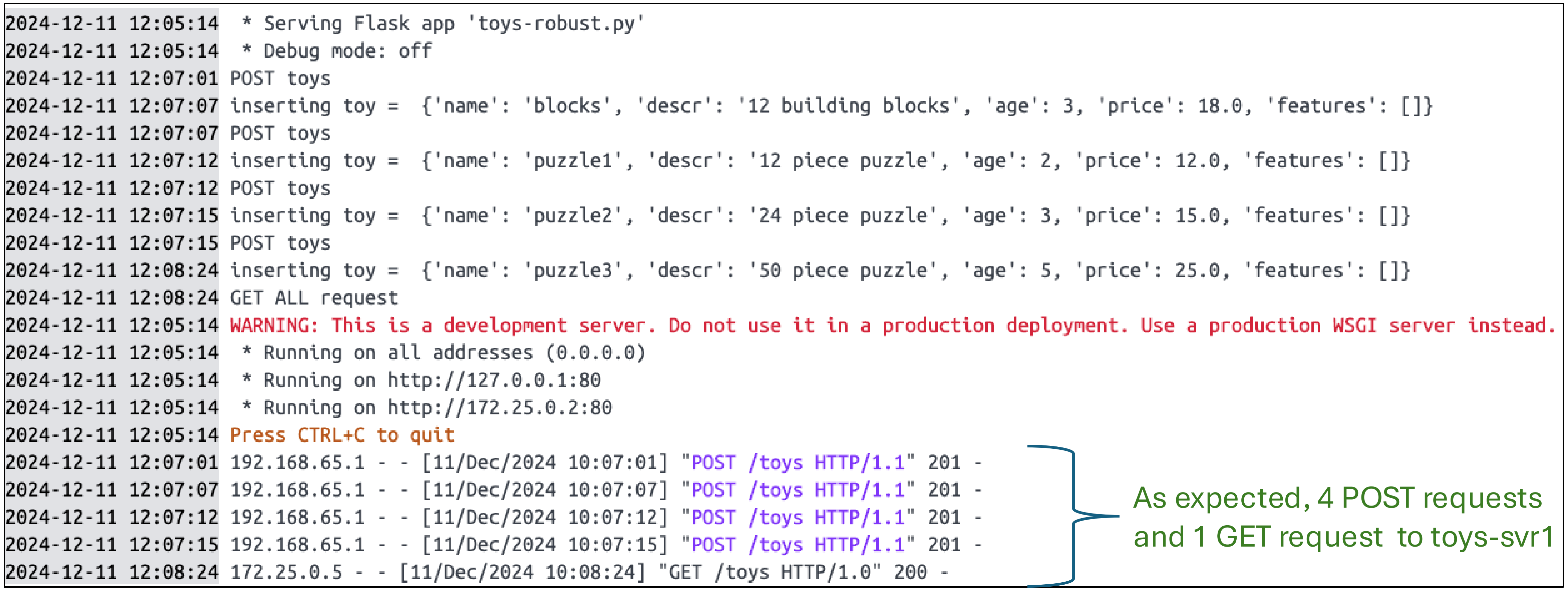


```
http {
    upstream back_end {
    server toys-svr1:80 weight=1;
    server toys-svr2:80 weight=1;
    }

    server {
    listen 80;
    location /toys {
        proxy_pass http://back_end;
        limit_except GET {  # allow
            deny all;
        }
    }
```

Trying it out

1. We first issue 4 POST requests to the toys service running at port 8001 and 1 POST request to the toys service running at port 8002.
2. We then issue 2 GET requests to port 80 (NGINX).

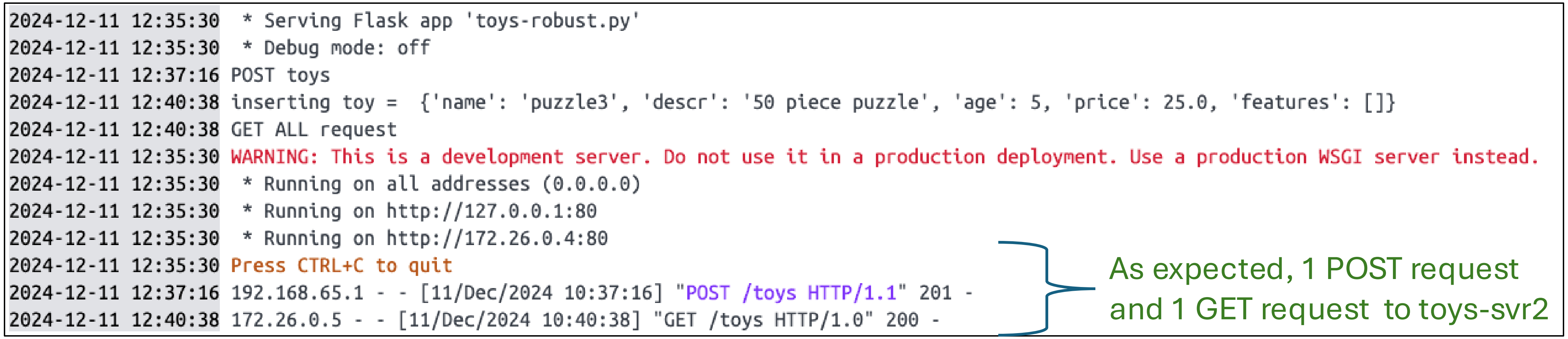
Log file for toys-svr1



```
2024-12-11 12:05:14  * Serving Flask app 'toys-robust.py'
2024-12-11 12:05:14  * Debug mode: off
2024-12-11 12:07:01 POST toys
2024-12-11 12:07:07 inserting toy =  {'name': 'blocks', 'descr': '12 building blocks', 'age': 3, 'price': 18.0, 'features': []}
2024-12-11 12:07:07 POST toys
2024-12-11 12:07:12 inserting toy =  {'name': 'puzzle1', 'descr': '12 piece puzzle', 'age': 2, 'price': 12.0, 'features': []}
2024-12-11 12:07:12 POST toys
2024-12-11 12:07:15 inserting toy =  {'name': 'puzzle2', 'descr': '24 piece puzzle', 'age': 3, 'price': 15.0, 'features': []}
2024-12-11 12:07:15 POST toys
2024-12-11 12:08:24 inserting toy =  {'name': 'puzzle3', 'descr': '50 piece puzzle', 'age': 5, 'price': 25.0, 'features': []}
2024-12-11 12:08:24 GET ALL request
2024-12-11 12:05:14 WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
2024-12-11 12:05:14  * Running on all addresses (0.0.0.0)
2024-12-11 12:05:14  * Running on http://127.0.0.1:80
2024-12-11 12:05:14  * Running on http://172.25.0.2:80
2024-12-11 12:05:14 Press CTRL+C to quit
2024-12-11 12:07:01 192.168.65.1 - - [11/Dec/2024 10:07:01] "POST /toys HTTP/1.1" 201 -
2024-12-11 12:07:07 192.168.65.1 - - [11/Dec/2024 10:07:07] "POST /toys HTTP/1.1" 201 -
2024-12-11 12:07:12 192.168.65.1 - - [11/Dec/2024 10:07:12] "POST /toys HTTP/1.1" 201 -
2024-12-11 12:07:15 192.168.65.1 - - [11/Dec/2024 10:07:15] "POST /toys HTTP/1.1" 201 -
2024-12-11 12:08:24 172.25.0.5 - - [11/Dec/2024 10:08:24] "GET /toys HTTP/1.0" 200 -
```

As expected, 4 POST requests and 1 GET request to toys-svr1

Log file for toys-svr2



```
2024-12-11 12:35:30  * Serving Flask app 'toys-robust.py'
2024-12-11 12:35:30  * Debug mode: off
2024-12-11 12:37:16 POST toys
2024-12-11 12:40:38 inserting toy =  {'name': 'puzzle3', 'descr': '50 piece puzzle', 'age': 5, 'price': 25.0, 'features': []}
2024-12-11 12:40:38 GET ALL request
2024-12-11 12:35:30 WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
2024-12-11 12:35:30  * Running on all addresses (0.0.0.0)
2024-12-11 12:35:30  * Running on http://127.0.0.1:80
2024-12-11 12:35:30  * Running on http://172.26.0.4:80
2024-12-11 12:35:30 Press CTRL+C to quit
2024-12-11 12:37:16 192.168.65.1 - - [11/Dec/2024 10:37:16] "POST /toys HTTP/1.1" 201 -
2024-12-11 12:40:38 172.26.0.5 - - [11/Dec/2024 10:40:38] "GET /toys HTTP/1.0" 200 -
```

$\}$ As expected, 1 POST request and 1 GET request to toys-svr2

Load balancing policies

- **Round Robin**. This is the default. It will forward to next server in the list and then wrap around.
- **Weighted Round Robin**. Each server is assigned a weight based on its processing capacity. Servers with higher weights receive more requests than those with lower weights.

```
upstream back_end {
server 172.17.0.1:8001 weight = 2;
server 172.17.0.1:8002 weight = 1;
}
```

8001 gets double the number of requests as 8002

See this article for a good overview of nginx load balancing policies

Other policies include:

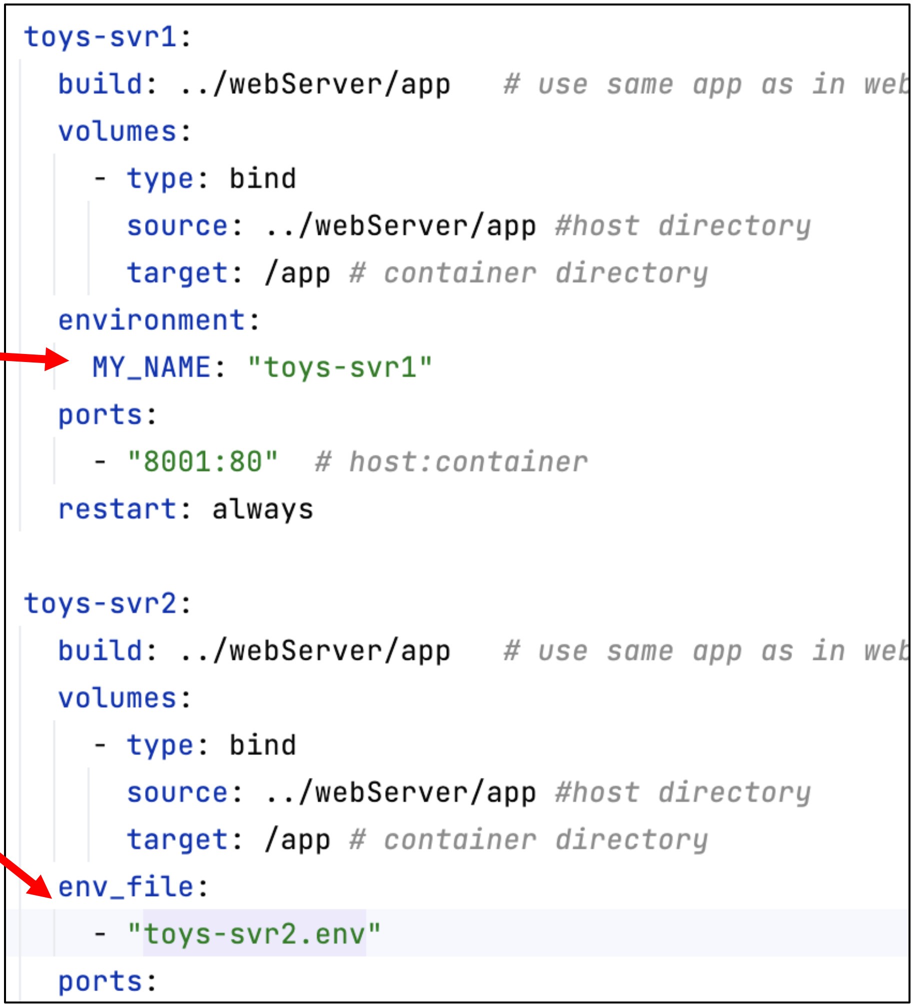
- **Least Connection**: NGINX routes to the server with the smaller number of current active connections.
- **Least Time**: NGINX selects the server with the lowest average latency and the least number of active connections.
- **IP-based hashing**: the clients IP address is used to determine which server to route to.
- **Path-based distribution**: the path of the incoming request (e.g, /resource1 versus /resource2) determines which server or service will handle the request.

Passing environment variables into a container

2 ways to pass in environment variables:

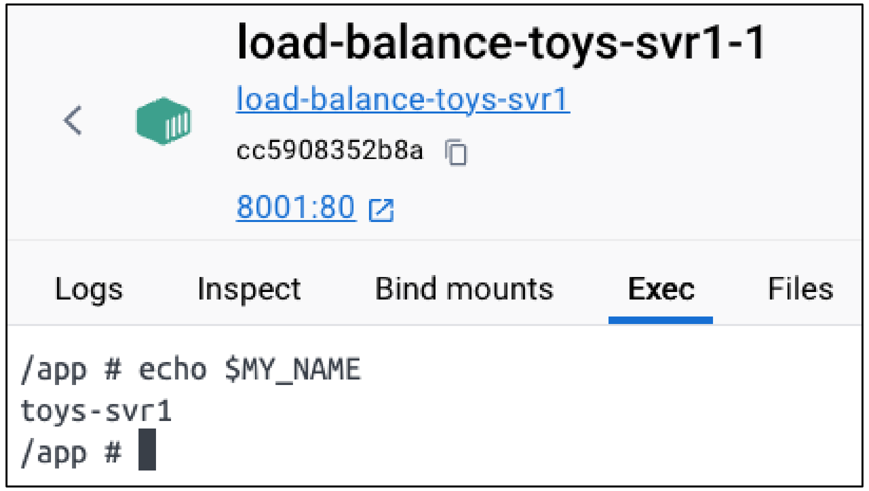
- Using the **environment** command. We set a single variable MY_NAME to be "toys-svr1".
- Using the **env_file** command. In this case, "toys-sv2.env" on the host contains a single line

```
MY_NAME=toys2
```

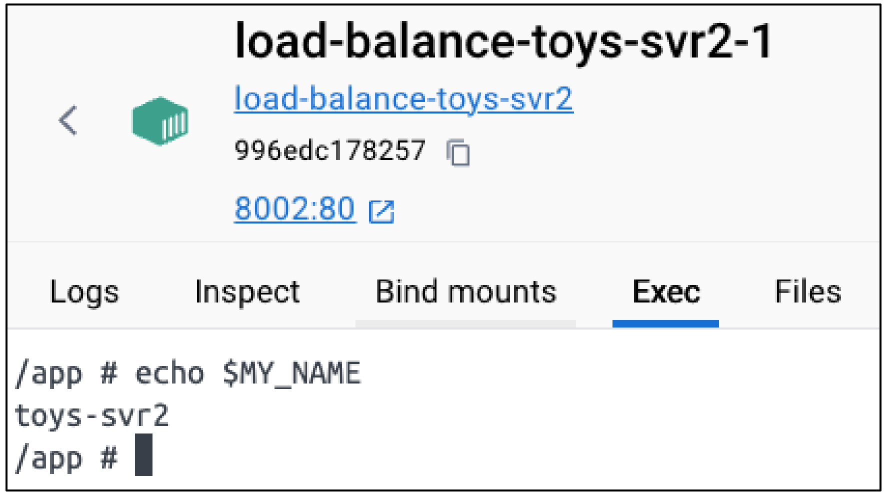


```
toys-svr1:
  build: ../webServer/app    # use same app as in web
  volumes:
    - type: bind
      source: ../webServer/app #host directory
      target: /app # container directory
  environment:
    MY_NAME: "toys-svr1"
  ports:
    - "8001:80"   # host:container
  restart: always

toys-svr2:
  build: ../webServer/app    # use same app as in web
  volumes:
    - type: bind
      source: ../webServer/app #host directory
      target: /app # container directory
  env_file:
    - "toys-svr2.env"
  ports:
```

Trying it out



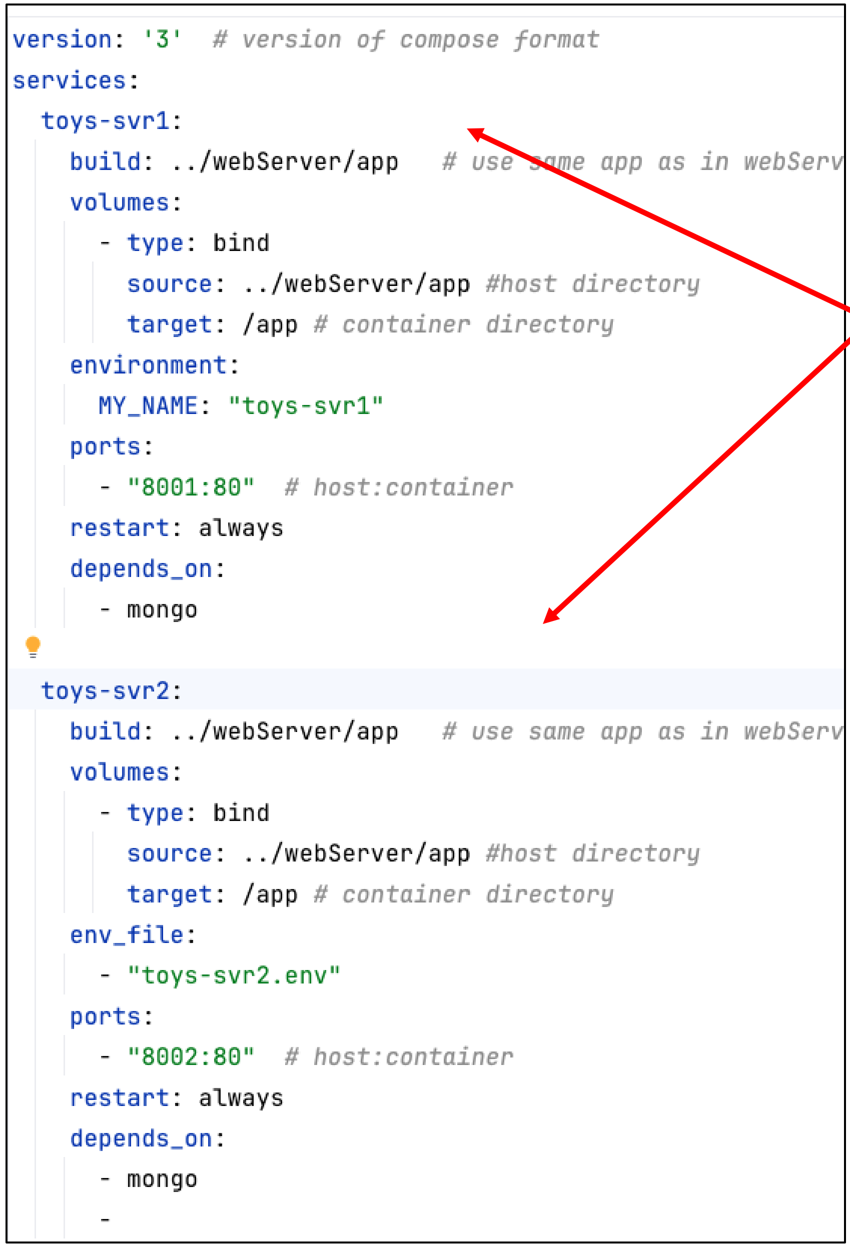
```
/app # echo $MY_NAME
toys-svr1
/app #
```



```
/app # echo $MY_NAME
toys-svr2
/app #
```

In Docker Compose Desktop enter into each container via the exec command.   Each container has a different setting for MY_NAME.

docker-compose-lb.yml

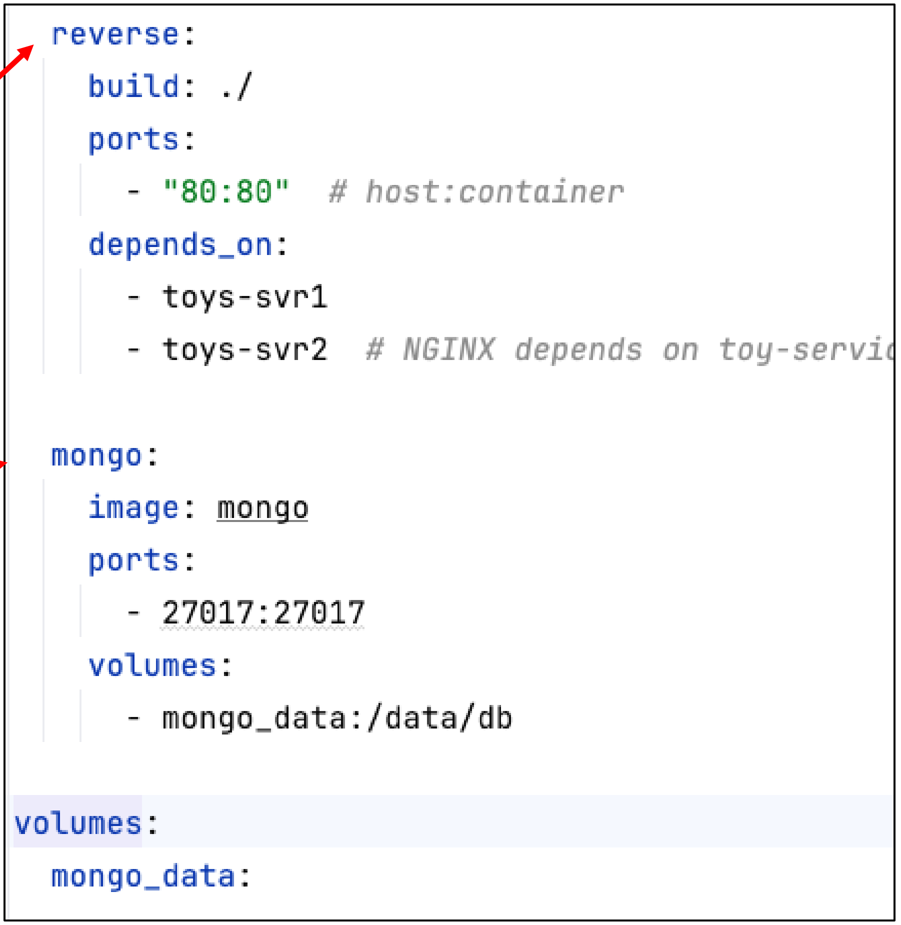


```
version: '3'  # version of compose format
services:
  toys-svr1:
    build: ../webServer/app    # use same app as in webServ
    volumes:
      - type: bind
        source: ../webServer/app #host directory
        target: /app # container directory
    environment:
      MY_NAME: "toys-svr1"
    ports:
      - "8001:80"  # host:container
    restart: always
    depends_on:
      - mongo

  toys-svr2:
    build: ../webServer/app    # use same app as in webServ
    volumes:
      - type: bind
        source: ../webServer/app #host directory
        target: /app # container directory
    env_file:
      - "toys-svr2.env"
    ports:
      - "8002:80"  # host:container
    restart: always
    depends_on:
      - mongo
      -
```

2 instances of toys svc.

NGINX reverse proxy

mongo

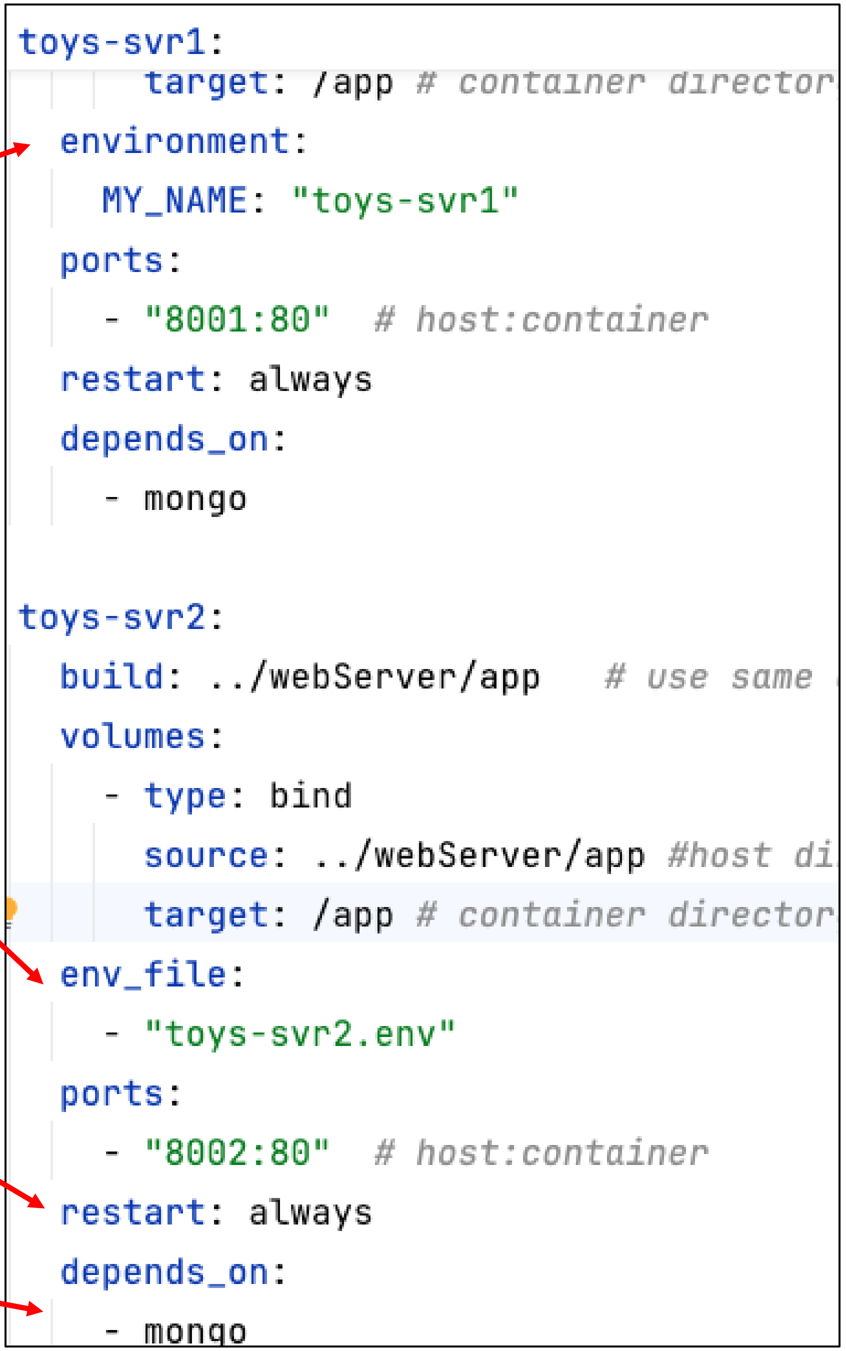


```
  reverse:
    build: ./
    ports:
      - "80:80"  # host:container
    depends_on:
      - toys-svr1
      - toys-svr2  # NGINX depends on toy-servic

  mongo:
    image: mongo
    ports:
      - 27017:27017
    volumes:
      - mongo_data:/data/db

volumes:
  mongo_data:
```

Two different ways of passing environment variables into a service

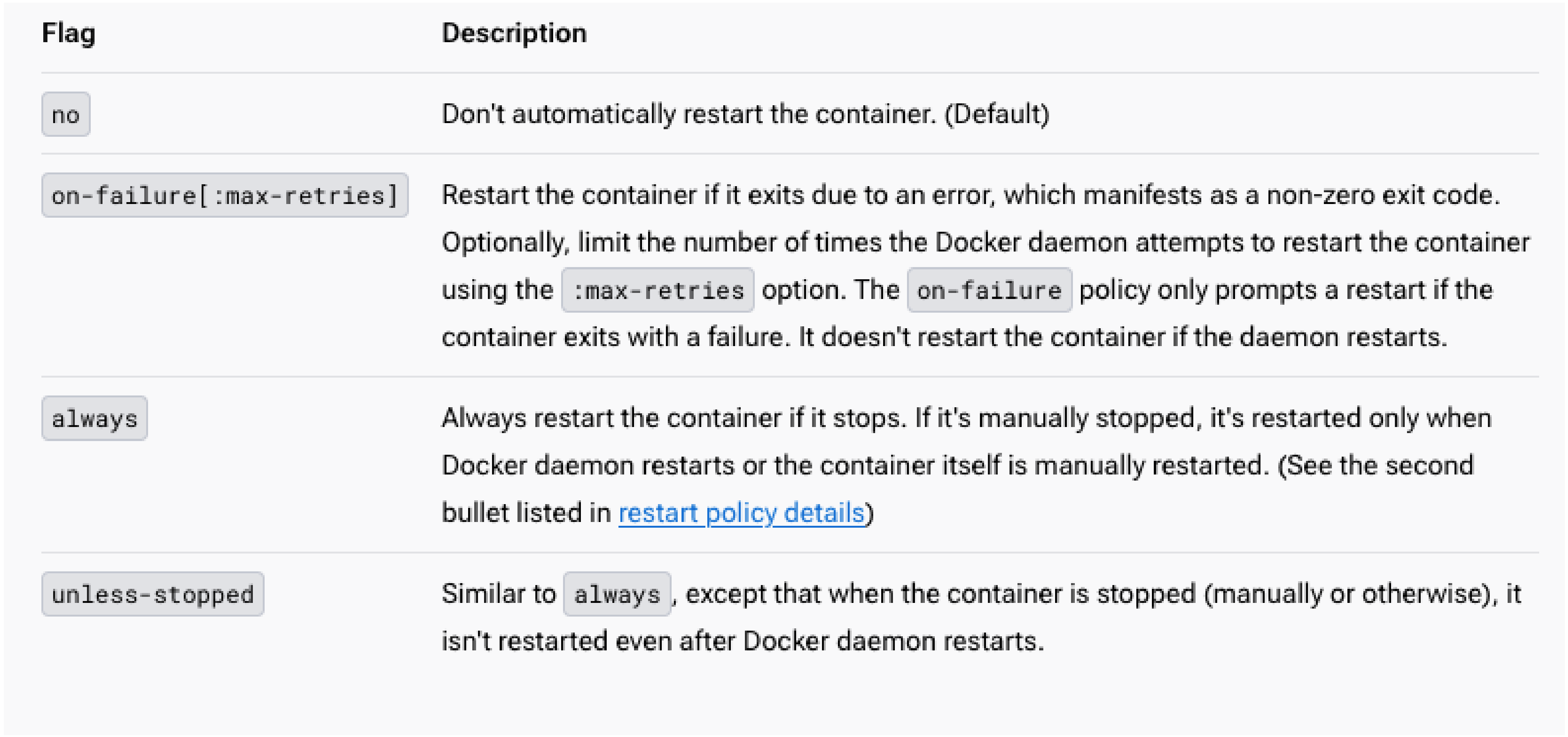
“restart: always” tells Docker to always restart the container when it dies, unless it is stopped manually.

Start mongo first and then the application.



```
toys-svr1:
      target: /app # container director
  environment:
    MY_NAME: "toys-svr1"
  ports:
    - "8001:80"   # host:container
  restart: always
  depends_on:
    - mongo

toys-svr2:
  build: ../webServer/app   # use same
  volumes:
    - type: bind
      source: ../webServer/app #host di
      target: /app # container director
  env_file:
    - "toys-svr2.env"
  ports:
    - "8002:80"   # host:container
  restart: always
  depends_on:
    - mongo
```

Docker-compose restart policies



Flag	Description
`no`	Don't automatically restart the container. (Default)
`on-failure[:max-retries]`	Restart the container if it exits due to an error, which manifests as a non-zero exit code. Optionally, limit the number of times the Docker daemon attempts to restart the container using the `:max-retries` option. The `on-failure` policy only prompts a restart if the container exits with a failure. It doesn't restart the container if the daemon restarts.
`always`	Always restart the container if it stops. If it's manually stopped, it's restarted only when Docker daemon restarts or the container itself is manually restarted. (See the second bullet listed in restart policy details)
`unless-stopped`	Similar to `always`, except that when the container is stopped (manually or otherwise), it isn't restarted even after Docker daemon restarts.