Install microservices
Before we start this scenario, we need to deploy all microservices (customer, preference, recommendation[v1:v2]).

There's a script called install-microservices.sh that will

- Checkout the source code from https://github.com/redhat-developer-demos/istio-tutorial
- Create recommendation:v2 with a bad performance (replies in 3 seconds).
- Run mvn package on all projects
- Create a docker image
- Deploy the microservices with the sidecar proxy

Execute this script: ./install-microservices.sh
The script will take between 2-5 minutes to complete. Don't worry if you see error messages.

When the scripts ends, watch the creation of the pods, execute oc get pods -w

Once that the microservices pods READY column are 2/2, you can hit CTRL+C.

Try the microservice by typing curl http://customer-tutorial.2886795302-80-frugo02.environments.katacoda.com

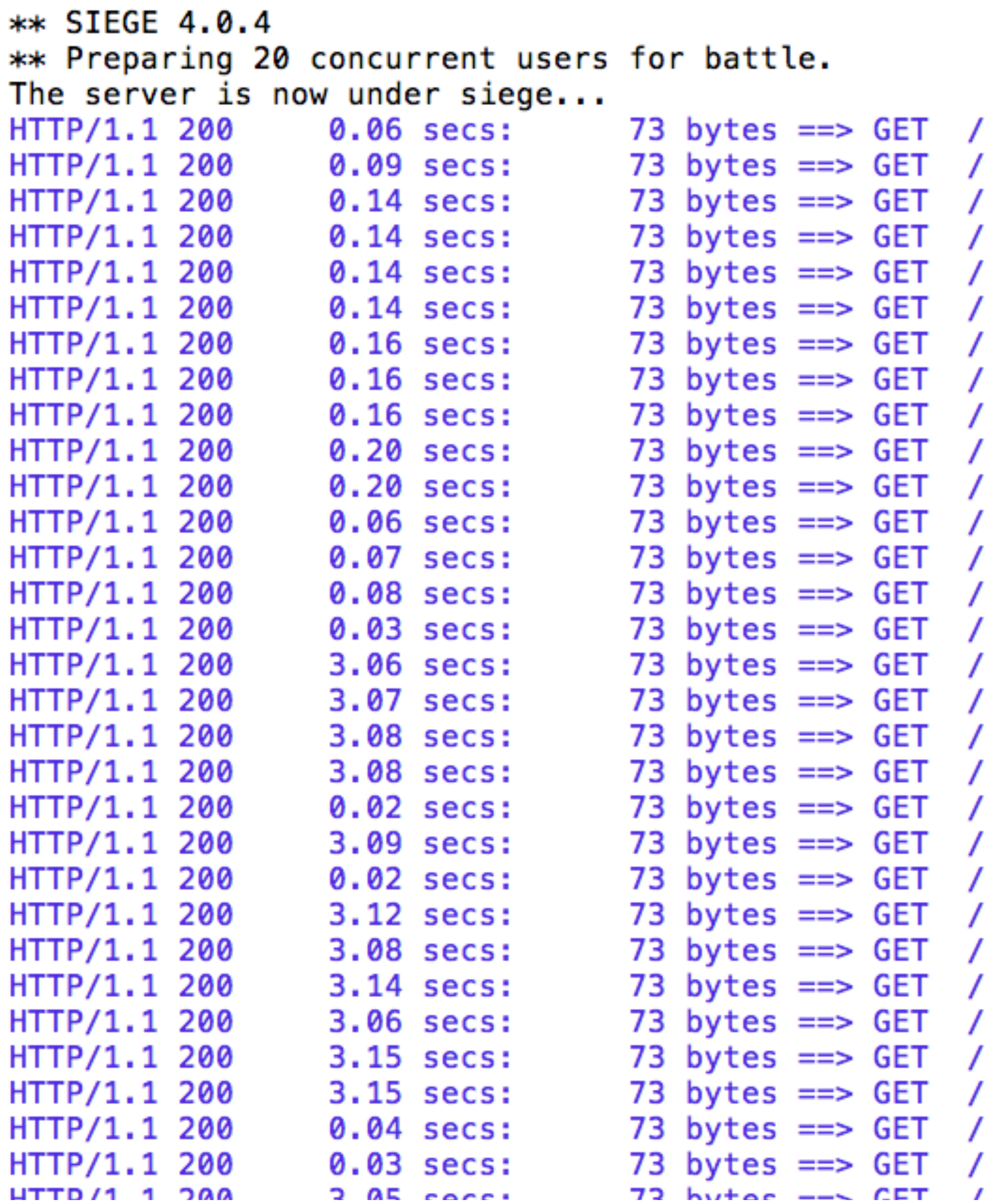
It should return:
customer => preference => recommendation v1 from {hostname}: 1

Fail Fast with Max Connections & Max Pending Requests
First, you need to insure you have a destinationrule and virtualservice in place. Let's use a 50/50 split of traffic:
Execute istioctl create -f ~/projects/istio-tutorial/istiofiles/destination-rule-recommendation-v1-v2.yml -n tutorial; \ istioctl create -f ~/projects/istio-tutorial/

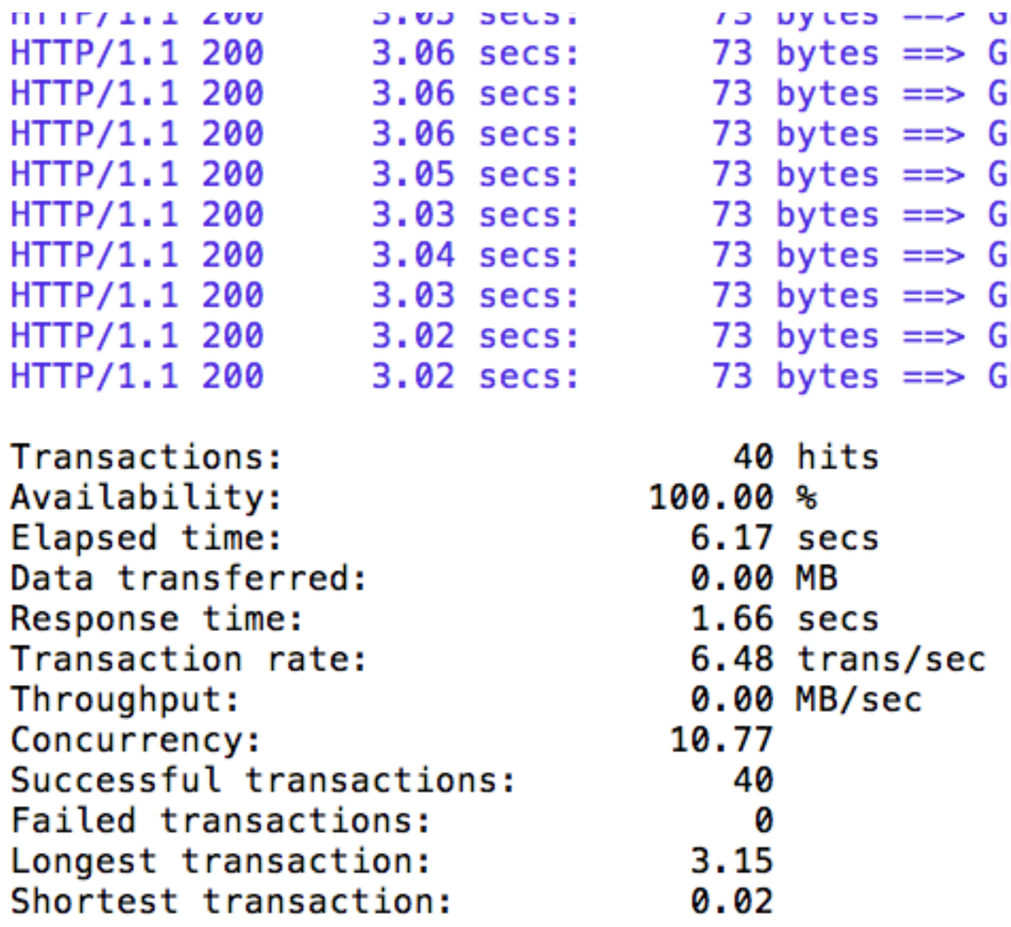
istiofiles/virtual-service-recommendation-v1_and_v2_50_50.yml -n tutorial

Load test without circuit breaker

Let's perform a load test in our system with siege. We'll have 20 clients sending 2 concurrent requests each:
siege -r 2 -c 20 -v http://customer-tutorial.2886795302-80-frugo02.environments.katacoda.com
You should see an output similar to this:



```
** SIEGE 4.0.4
** Preparing 20 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200     0.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.09 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.14 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.14 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.14 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.14 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.16 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.16 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.16 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.20 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.20 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.07 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.08 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.03 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.07 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.08 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.08 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.02 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.09 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.02 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.12 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.08 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.14 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.15 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.15 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.04 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.03 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.05 secs:      73 bytes ==> GET  /
```



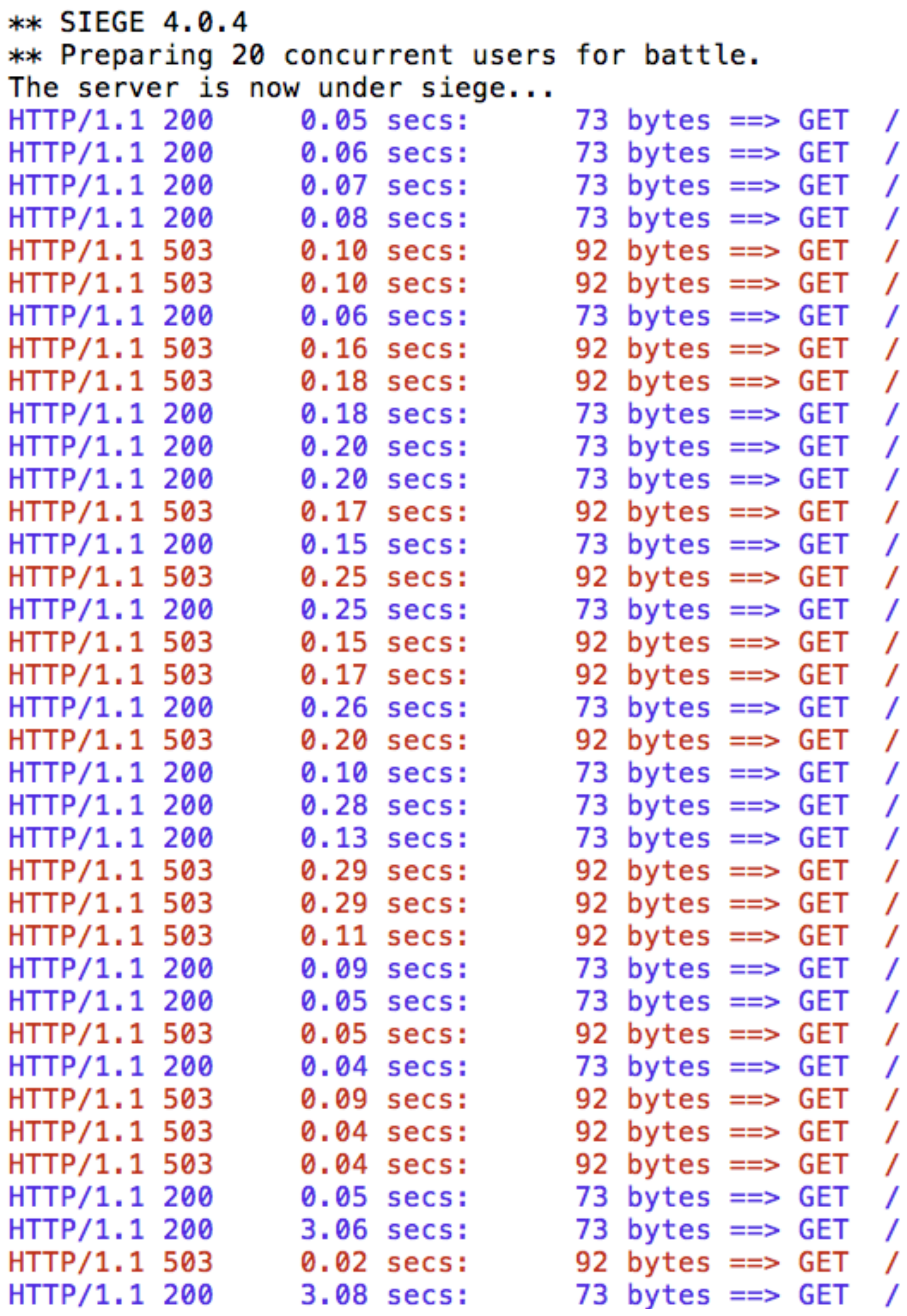
```
HTTP/1.1 200     3.05 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.05 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.03 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.04 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.03 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.02 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.02 secs:      73 bytes ==> GET  /

Transactions:                     40 hits
Availability:                 100.00 %
Elapsed time:                   6.17 secs
Data transferred:               0.00 MB
Response time:                  1.66 secs
Transaction rate:               6.48 trans/sec
Throughput:                     0.00 MB/sec
Concurrency:                   10.77
Successful transactions:          40
Failed transactions:               0
Longest transaction:            3.15
Shortest transaction:           0.02
```

All of the requests to our system were successful, but it took some time to run the test, as the v2 instance/pod was a slow performer.
But suppose that in a production system this 3s delay was caused by too many concurrent requests to the same instance/pod. We don't want multiple requests getting queued or making the instance/pod even slower. So we'll add a circuit breaker that will **open** whenever we have more than 1 request being handled by any instance/pod.

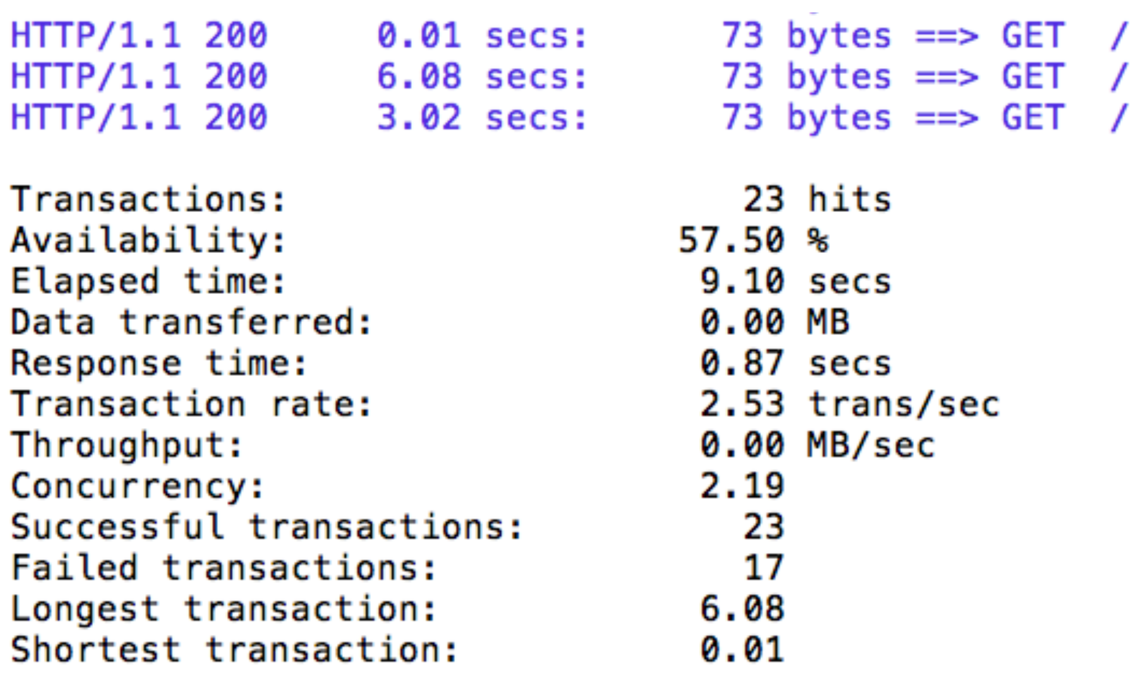
Check the file /istiofiles/destination-rule-recommendation_cb_policy_version_v2.yml.
Let's apply this DestinationRule: istioctl replace -f ~/projects/istio-tutorial/istiofiles/destination-rule-recommendation_cb_policy_version_v2.yml -n tutorial
More information on the fields for the simple circuit-breaker https://istio.io/docs/reference/config/istio.networking.v1alpha3/#OutlierDetection

Load test with circuit breaker

Now let's see what is the behavior of the system running siege again:
siege -r 2 -c 20 -v http://customer-tutorial.2886795302-80-frugo02.environments.katacoda.com
You should see an output similar to this:



```
** SIEGE 4.0.4
** Preparing 20 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200     0.05 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.07 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.08 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.10 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.10 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.06 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.16 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.18 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.18 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.20 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.20 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.17 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.15 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.25 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.25 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.15 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.17 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.26 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.20 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.10 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.28 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.13 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.29 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.29 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.11 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.09 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.05 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.05 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.04 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.09 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.04 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.04 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.05 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.06 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.02 secs:      92 bytes ==> GET  /
HTTP/1.1 200     3.08 secs:      73 bytes ==> GET  /
```



```
HTTP/1.1 200     0.01 secs:      73 bytes ==> GET  /
HTTP/1.1 200     6.08 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.02 secs:      73 bytes ==> GET  /

Transactions:                      23 hits
Availability:                   57.50 %
Elapsed time:                    9.10 secs
Data transferred:                0.00 MB
Response time:                   0.87 secs
Transaction rate:                2.53 trans/sec
Throughput:                      0.00 MB/sec
Concurrency:                     2.19
Successful transactions:           23
Failed transactions:               17
Longest transaction:             6.08
Shortest transaction:            0.01
```

You can run siege multiple times, but in all of the executions you should see some 503 errors being displayed in the results. That's the circuit breaker being opened whenever Istio detects more than 1 pending request being handled by the instance/pod.

Clean up

Don't forget to remove the virtualservice and destinationrule executing ~/projects/istio-tutorial/scripts/clean.sh

Pool Ejection

Pool ejection or *outlier detection* is a resilience strategy that takes place whenever we have a pool of instances/pods to serve a client request. If the request is forwarded to a certain instance and it fails (e.g. returns a 50x error code), then Istio will eject this instance from the pool for a certain *sleep window*. In our example the sleep window is configured to be 15s. This increases the overall availability by making sure that only healthy pods participate in the pool of instances.
First, you need to insure you have a destinationrule and virtualservice in place. Let's use a 50/50 split of traffic:

istioctl create -f ~/projects/istio-tutorial/istiofiles/destination-rule-recommendation-v1-v2.yml -n tutorial; \ istioctl create -f ~/projects/istio-tutorial/

istiofiles/virtual-service-recommendation-v1_and_v2_50_50.yml -n tutorial

Scale number of instances of v2 deployment

oc scale deployment recommendation-v2 --replicas=2 -n tutorial

Execute oc get pods -w

Once that the microservices pods READY column are 2/2, you can hit CTRL+C.

Test behavior without failing instances

Execute on Terminal 2 while true; do curl http://customer-tutorial.2886795302-80-frugo02.environments.katacoda.com; sleep .1; done

You will see the load balancing 50/50 between the two different versions of the recommendation service. And within version v2, you will also see that some requests are handled by one pod and some requests are handled by the other pod.

```
customer => preference => recommendation v1 from '2039379827-jmm6x': 447
customer => preference => recommendation v2 from '2036617847-spdrb': 26
customer => preference => recommendation v1 from '2039379827-jmm6x': 448
customer => preference => recommendation v2 from '2036617847-spdrb': 27
customer => preference => recommendation v1 from '2039379827-jmm6x': 449
customer => preference => recommendation v1 from '2039379827-jmm6x': 450
```

Test behavior with failing instance and without pool ejection

Let's get the name of the pods from recommendation v2:

oc get pods -l app=recommendation,version=v2

You should see something like this:

```
recommendation-v2-2036617847-hdjv2   2/2      Running   0         1h
```

```
recommendation-v2-2036617847-spdrb   2/2       Running   0          7m
```

Now we'll get into one the pods and add some erratic behavior on it.

```
oc exec -it $(oc get pods|grep recommendation-v2|awk '{ print $1 }'|head -1) -c recommendation /bin/bash
```

You will be inside the application container of your pod. Now execute the following command inside the recommenation-v2 pod:

```
curl localhost:8080/misbehave
```

Now exit from the recommendation-v2 pod:

```
exit
```

This is a special endpoint that will make our application return only 503s.

Make sure that the following command is running on Terminal 2 while true; do curl http://customer-tutorial.2886795302-80-frugo02.environments.katacoda.com; sleep .1; done

You'll see that whenever the pod that you ran the command curl localhost:8080/misbehave receives a request, you get a 503 error:

```
customer => preference => recommendation v2 from '2036617847-hdjv2': 248
customer => preference => recommendation v1 from '2039379827-jmm6x': 496
customer => preference => recommendation v1 from '2039379827-jmm6x': 497
customer => 503 preference => 503 recommendation misbehavior from '2036617847-spdrb'
customer => preference => recommendation v2 from '2036617847-hdjv2': 249
customer => preference => recommendation v1 from '2039379827-jmm6x': 498
customer => 503 preference => 503 recommendation misbehavior from '2036617847-spdrb'
customer => preference => recommendation v2 from '2036617847-hdjv2': 250
```

Test behavior with failing instance and with pool ejection

Now let's add the pool ejection behavior:

Check the file /istiofiles/destination-rule-recommendation_cb_policy_pool_ejection.yml.
Now execute:

```
istioctl replace -f ~/projects/istio-tutorial/istiofiles/destination-rule-recommendation_cb_policy_pool_ejection.yml -n tutorial
```

Make sure that the following command is running on Terminal 2 while true; do curl http://customer-tutorial.2886795302-80-frugo02.environments.katacoda.com; sleep .1; done

You will see that whenever you get a failing request with 503 from the pod, it gets ejected from the pool, and it doesn't receive any more requests until the sleep window expires - which takes at least 15s.

```
customer => preference => recommendation v1 from '2039379827-jmm6x': 509
customer => 503 preference => 503 recommendation misbehavior from '2036617847-spdrb'
customer => preference => recommendation v1 from '2039379827-jmm6x': 510
customer => preference => recommendation v1 from '2039379827-jmm6x': 511
customer => preference => recommendation v1 from '2039379827-jmm6x': 512
customer => preference => recommendation v1 from '2039379827-jmm6x': 513
customer => preference => recommendation v1 from '2039379827-jmm6x': 514
customer => preference => recommendation v2 from '2036617847-hdjv2': 256
customer => preference => recommendation v2 from '2036617847-hdjv2': 257
customer => preference => recommendation v1 from '2039379827-jmm6x': 515
customer => preference => recommendation v2 from '2036617847-hdjv2': 258
customer => preference => recommendation v2 from '2036617847-hdjv2': 259
customer => preference => recommendation v2 from '2036617847-hdjv2': 260
customer => preference => recommendation v1 from '2039379827-jmm6x': 516
customer => preference => recommendation v1 from '2039379827-jmm6x': 517
customer => preference => recommendation v1 from '2039379827-jmm6x': 518
customer => 503 preference => 503 recommendation misbehavior from '2036617847-spdrb'
customer => preference => recommendation v1 from '2039379827-jmm6x': 519
```

Hit CTRL+C when you are satisfied.

Ultimate resilience with retries, circuit breaker, and pool ejection
Even with pool ejection your application doesn't look that resilient. That's probably

because we're still letting some errors to be propagated to our clients. But we can improve this. If we have enough instances and/or versions of a specific service running into our system, we can combine multiple Istio capabilities to achieve the ultimate backend resilience:

- **Circuit Breaker** to avoid multiple concurrent requests to an instance;
- **Pool Ejection** to remove failing instances from the pool of responding instances;
- **Retries** to forward the request to another instance just in case we get an open circuit breaker and/or pool ejection;

By simply adding a **retry** configuration to our current routerule, we'll be able to get rid completely of our 503s requests. This means that whenever we receive a failed request from an ejected instance, Istio will forward the request to another supposably healthy instance.

Check the file /istiofiles/virtual-service-recommendation-v1_and_v2_retry.yml. Execute:

```
istioctl replace -f ~/projects/istio-tutorial/istiofiles/virtual-service-recommendation-v1_and_v2_retry.yml -n tutorial
```

Make sure that the following command is running on Terminal 2 while true; do curl http://customer-tutorial.2886795302-80-frugo02.environments.katacoda.com; sleep .2; done
You won't receive 503s anymore. But the requests from recommendation v2 are still taking more time to get a response. Our misbehaving pod never shows up in the console, thanks to pool ejection and retry.

Clean up

Reduce the number of v2 replicas to 1: oc scale deployment recommendation-v2 --replicas=1 -n tutorial
Delete the failing pod: oc delete pod -l app=recommendation,version=v2
Don't forget to remove the virtualservice and destinationrule executing ~/projects/istio-tutorial/scripts/clean.sh