# Application Generator:

The only purpose of the application generator is to generate the kubernetes manifest files. The command that will generate has following syntax.

***go run main.go generate <chain\_file\_name> <cluster\_file\_name> <readiness\_probe\_for\_k8s>***

Where chain\_file and cluster\_file are located under chains and clusters folder respectively inside the application generator project.

Chain\_file describes the chain of microservices and cluster\_file defines how the microservices are placed inside the cluster. Hence, the user should make appropriate changes before running this command. Both the files follow the JSON format

**Example chain file**

{

"serviceID": 1,

"chains": [

{

"latency": 0.045,

"chainID": "1",

"microServices": [

"productpage",

"details",

"reviews",

"ratings"

]

},

{

"latency": 0.090,

"chainID": "2",

"microServices": [

"reviews",

"ratings"

]

}

]

**Example cluster file**

{

"service\_id": "test",

"final\_placement": [

{

"cluster\_id": "cluster2",

"service\_list": [

"reviews",

"details",

"ratings",

"productpage"

]

},

{

"cluster\_id": "cluster3",

"service\_list": [

"reviews",

"details",

"productpage"

]

}

]

}

In the above example, the chain file defines two chains each having a chain ID. Each chain then includes a set of microservices defined in an order. The order in which the microservices are defined forms the chain of microservices.

In the cluster file, we can see how each microservice is places in the cluster. The file describes a cluster id for each cluster where we want to run our microservices. Each cluster id has a list of microservices defined. This list tells what microservices will be running inside that cluster. Here the order in which microservices are listed under cluster is not important.

The K8s\_readiness\_probe is used to simulate the startup time for microservices. This is a numeric value and is the time taken in seconds by Kubernetes to mark the pod as ready from the time the pod is created. For example, if readiness probe is set to 20, the Kubernetes will take 20 seconds to mark the pod for which readiness probe is defined, as ready after the pod was created. In the application generator, this probe is defined for the main application accepting traffic running inside the pod.

The go command described above requires certain go and cobra libraries which it fails to find unless the location of the project is under go/src/. Hence it is important that the user place application-generator project under go/src/ path. If the command still fails to find some packages, that means they are not present on user’s machine and can be downloaded using “go” command.

If the command runs successfully, two types of files will be generated. One under “*mutual*” folder that will include the service manifest files. These manifest files will be common across all clusters. Other under the “<cluster\_id>” folder which will contain the deployment files for the microservices in each cluster.

The deployment file contains deployment for the main application pod. The pod manifest has configurations for four containers, one for the main app the accepts the user traffic, second is the redis-database, third is for the worker container that will do the tasks assigned by the main application and fourth container is the where the fortio server is running.

## Added support

1. Added support for creation of additional container for fortio server while main app manifest files are generated.
2. Added fortio port in the service manifest files.
3. Integrated the main app and worker app into single deployment
4. Changed the namespace from default to edge-namespace

# Application Model

This is the main application that will run as a container inside the main app deployment. It serves two main tasks, one to listen to the incoming traffic and second to assign tasks to worker

## Added support

1. Updated the dockerfile to download fortio into the image that can be later used for load testing. Fortio will be used in the source microservice main app and worker to send the load to the fortio server running inside container in the destination microservice. The main application itself does not send traffic; it is the worker that will perform the actual task. The main application will only assign the task.
2. While performing the load test, the user can now specify the delay in response, response size, response status and other fortio server related configurations.

## Application Model Worker

This is the worker application that will perform tasks assigned by the main application by reading them from the redis queue.

## Added support

1. Updated the dockerfile to download fortio into the image that can be later used for load testing. Fortio will be used in the source microservice main app and worker to send the load to the fortio server running inside container in the destination microservice. The main application itself does not send traffic, it is the worker that will perform the actual task. The main application will only assign the task.

# Steps to run the test

Expose the microservice deployment as LoadBalancer, which will be the first in chain

Tsung tool will be used to run the load test.

Move to tsung folder inside the application-generator directory.

Update the chain.json file and a possible valid file might look like this

{

"initial": "s3",

"chain\_no": "2",

"request\_task\_type": {

"1": "cpu",

"2": {"communication": {"client\_params": "-qps 1000 -c 8 -t 3s -r 0.0001",

"server\_params": "delay=0.5s:50,1s:40&size=512:10,1024:90"}},

"3": "sleep",

"4": "memory",

"5": {"communication": {"client\_params": "-qps 1000 -c 8 -t 3s -r 0.0001",

"server\_params": "delay=0.5s:50,1s:40&size=512:10,1024:90"}},

"6": "sleep",

"7": "cpu",

"8": {"communication": {"client\_params": "-qps 1000 -c 8 -t 3s -r 0.0001",

"server\_params": "delay=0.5s:50,1s:40&size=512:10,1024:90"}},

"9": "memory"

},

"request\_type": "1"

}

Where:

**initial** is the first microservice in the chain

**chain\_no** is the chain that we want to follow, this chain\_no is found inside the chain file used while generating manifest files

**request\_task\_type** is the type of task the user wants to be done from microservices. In the given example, there were 9 microservices running in the chain, and each microservice is given a task. Task can be of four types: “**cpu**”, “**sleep**”,“**memory**” or“**communication**”.

Further if the communication type of task is requested from the microservice, the user also needs to provide the parameters for client side “**client\_params**” and for server side “**server\_params**” the value of which will be passed to fortio while load testing. In the given example we are telling fortio that that from client side, It should make 1000 queries per second using 8 threads(connections) for 3 seconds. From server side while getting response from it, the user wants 50% of the requests to have a delay of 0.5 s in the response and 40% will have a delay of 1s. Similarly, the probabilities can be mentioned for response sizes. The probabilities do not necessarily need to be add up to 100%, the user can mention any probabilities, rest of the requests will have the default behavior set by fortio. More configurations can be set by the user and details can be found here

[https://github.com/fortio/fortio#server-urls-and-features](https://github.com/fortio/fortio" \l "server-urls-and-features)

If user does not want to mention the client\_params or server\_params, then he/she can leave the value field blank by providing simple quotes (“”).

If the user mentions the lesser number of tasks\_type in the files than there are microservices in the chain, then rest of the microservices will perform a random task between **cpu, memory or sleep**.

**request\_type**: this the task\_type that will be performed by the first microservice. It should have a value of 1. In the above example, 9 tasks\_type are defined. Mentioning the value of 1 in request\_type tells the first microservice in the chain to do the task 1 (cpu in the example). The following microservices in the chain will perform the next tasks in order specified in the task\_type. So, for example, microservice 2 will perform communication task, microservice 3 will perform sleep task and so on. User can specify the request\_type to any other number as well (for example 3) then the first microservice in chain will perform task 3 and the other microservices will follow this.

Next update the **conf.xml** file under tsung directory and update the **host** entry under **<servers>** tag. Update the Load Balancer address of the first microservice that we exposed in the last step. Also update the port if required.

Also update the **contents\_from\_file** entry under **<sessions>** tag and specify the name of conf file that was updated in the previous step.

Command: ***tsung -f conf.xml -k start***

## Introducing network delays using Pumba

*pumba netem --duration 5m --target* ***target\_IP*** *delay --time* ***time\_ms*** *containers* ***source\_container***

where

target\_IP is the IP of the istio ingress gateway of the target cluster container whose incoming traffic from source\_containerwill have the specified latency

time\_ms is the latency time in milliseconds

source\_container is the name of the source container whose outgoing traffic to target\_IP will have the specified latency

duration is the time for which the network latency will be introduced

Note: If target\_IP is not specified, all the outgoing traffic from source\_container will have the specified latencty