

Genesys Universal Messaging

Gildas Cherruel (gildas.cherruel@genesys.com)

v. 1.0.4 (Chart: 1.0.4), 2022/04/29

Contents

1 Concepts	3
2 How to Install/deploy Universal Messaging on Kubernetes	7
Pre-requisites	7
Installing Universal Messaging	7
The automated way	7
The manual way	9
Installing the beta/Development versions	10
Upgrading Universal Messaging	10
Fine tuning	11
Deploying only the services you need	11
Extra logging	11
Horizontal Auto-Scaling	12
Use a Custom Container Registry	12
Using external Redis or RabbitMQ	12
Resource limits and requests	13
Configuring HTTPS	15
Getting an FQDN with AWS	15
Getting an FQDN with Azure	16
Getting an FQDN with Google Cloud Platform	17
Logging Configuration	17
Tips and Tricks for Checking the deployment	17
Access the Redis Database	17
Backing up the Redis Database	18
Access the RabbitMQ Dashboard	19
3 Configuration	21
Common Settings	21
Notifier Destinations	21
Commanders	28
Storages	34
Tenants	35
CX	36
Messaging	43
Viber	46
Changing the administrator password	47
4 Using the REST API	49
REST API for IVRs	49
5 Kubernetes Fundamentals	51
Google Cloud Platform (GKE)	51
Microsoft Azure Container Service (AKS)	51
Amazon Elastic Container Services for Kubernetes (EKS)	53
Fargate	53
AWS EFS CSI Driver for Persistent Volumes	54
Deploy the ALB Ingress Controller	56
Delete the cluster	58
Docker for Desktop	58
MicroK8s	59

Manual Deployment on Virtual Machines/Bare Metal	59
6 Getting Helm	61
7 Troubleshooting	63
Helm upgrade is stuck	63

Chapter 1

Concepts

Genesys Universal Messaging is a collection of micro-services connected with each other via message queueing and managed by Kubernetes.

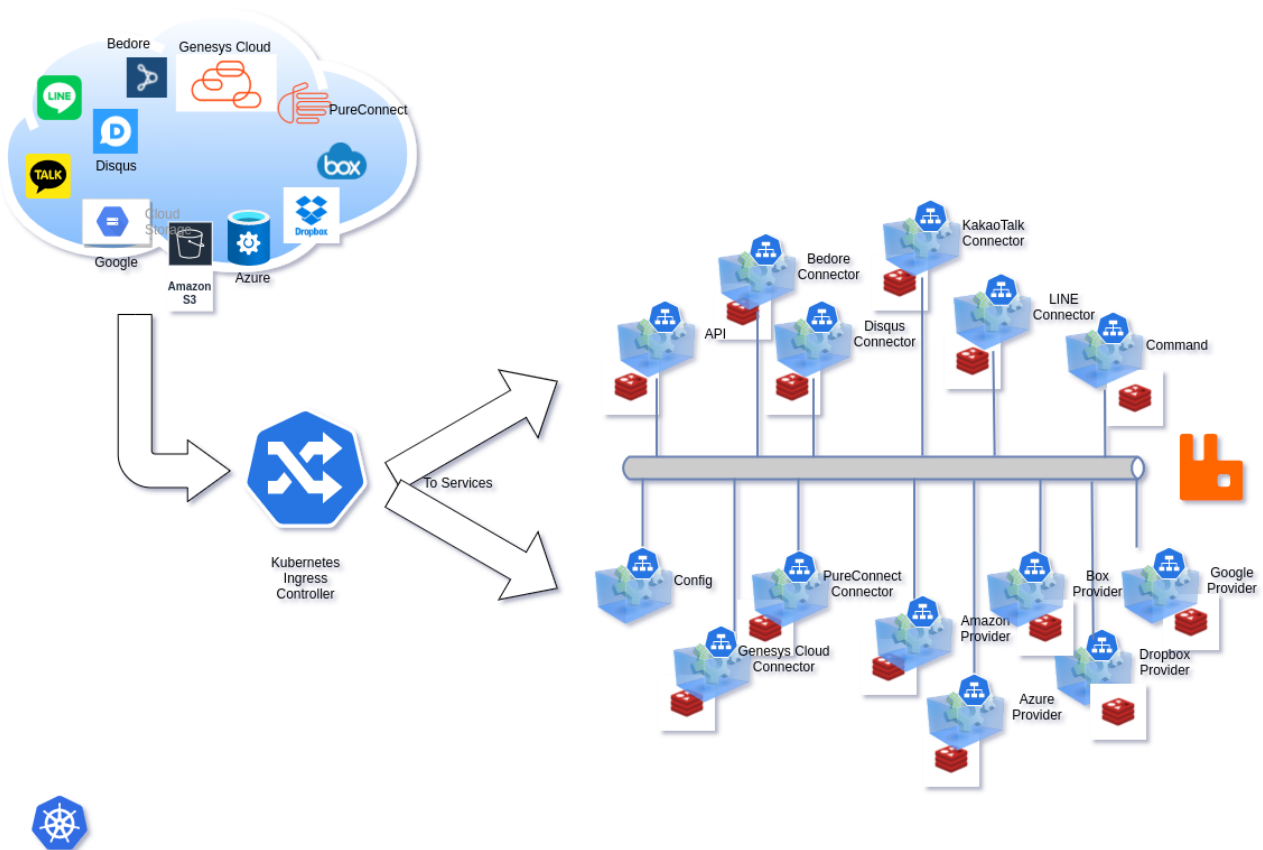
Each micro-service is responsible to connect to one service that will provide either a Social Media, a Customer Center Platform, or a Storage. Some micro-services will, on the other hand, provide a service of their own, like the API, the config website.

Here is a non-exhaustive list of these micro-services:

- API, provides the REST API to configure and use Universal Messaging;
- Config, provides a website to configure Universal Messaging;
- Commander, provides commands to the agent (see later);
- Apple Messages for Business Connector, interfaces with [Apple Messages for Business](#)
- Bedore Connector, interfaces with the Bedore Bot;
- BizM for KakaoTalk, interfaces with the [KakaoTalk](#) Social Media via [BizM](#);
- Disqus Connector, interfaces with [Disqus](#) Comment Service;
- Google Business Messages Connector, interfaces with [Google Business Messages](#);
- Infobank for KakaoTalk, interfaces with the [KakaoTalk](#) Social Media via [Infobank](#);
- LINE Connector, interfaces with the [LINE](#) Social Media;
- Media4U Connector, interfaces with the [Media4U](#) SMS Services;
- PlusMessage Connector, interfaces with [DOCOMO +Message](#) Social Media;
- Telegram Connector, interfaces with the [Telegram](#) Social Media;
- Viber Connector, interfaces with the [Viber](#) Social Media;
- WeChat Connector, interfaces with the [WeChat](#) Social Media;
- Genesys Cloud CX, the [Genesys Cloud CX](#) platform;
- PureConnect, the [Genesys PureConnect CX](#) platform;
- AWS Provider, [Amazon Web Services S3 Storage](#);
- Azure Provider, [Microsoft Azure Blob Storage](#);
- Box Provider, [Box.com](#);
- Dropbox Provider, [Dropbox](#);
- Google Provider, [Google Cloud Storage](#);

All these services are connected together via the [RabbitMQ](#) Message Queueing System. It is used to exchange messages from a Social Media and a CX platform.

The following diagram describes all micro-services in the context of their relation with the Social Media, CX platforms, and Storage providers.



Each micro-service is a collection of Kubernetes Deployments, Services, like:

- a ClusterIP Kubernetes service,
- a Deployment,
- a REDIS (service + Stateful)

Only the Config micro-service does not carry a REDIS.

The micro-services access directly the Social Media, CX Platform, or Storage Provider they support. It is possible to make them access through a Proxy by using the config website.

A Kubernetes Ingress controller is needed to send the traffic to the respective micro-services. The Ingress Object looks like (not all properties are show):

```

apiVersion: networking.k8s.io/v1
kind: Ingress
spec:
  rules:
  - http:
      paths:
      - path: /api
        backend:
          service:
            name: XX-api
            port: { number: 80 }
      - backend:
          service:
            name: XX-amb-connector
            port: { number: 80 }
        path: /amb
      - backend:
          service:
            name: XX-bedore-connector
            port: { number: 80 }
        path: /bedore
      - backend:

```

```
    service:
      name: XX-bizmsg-connector
      port: { number: 80 }
  path: /bizmsg
- backend:
  service:
    name: XX-disqus-connector
    port: { number: 80 }
  path: /disqus
- backend:
  service:
    name: XX-gbm-connector
    port: { number: 80 }
  path: /gbm
- backend:
  service:
    name: XX-infobank-connector
    port: { number: 80 }
  path: /infobank
- backend:
  service:
    name: XX-line-connector
    port: { number: 80 }
  path: /line
- backend:
  service:
    name: XX-media4u-connector
    port: { number: 80 }
  path: /media4u
- backend:
  service:
    name: XX-plusmessage-connector
    port: { number: 80 }
  path: /plusmessage
- backend:
  service:
    name: XX-telegram-connector
    port: { number: 80 }
  path: /telegram
- backend:
  service:
    name: XX-viber-connector
    port: { number: 80 }
  path: /viber
- backend:
  service:
    name: XX-wechat-connector
    port: { number: 80 }
  path: /wechat
- backend:
  service:
    name: XX-gcloudcx-connector
    port: { number: 80 }
  path: /openmessaging
- path: /
  backend:
    service:
      name: XX-config
      port: { number: 80 }
```

Notes:

- The TLS configuration is not included in this YAML.
- XX should be replaced by the Helm Release name.

Chapter 2

How to Install/deploy Universal Messaging on Kubernetes

Pre-requisites

As obvious as this may be, you must have a Kubernetes cluster (version 1.18+) to be able to deploy this application. It has to be production ready. We show [here](#) how to get one in the most common environments.

You also must have [Helm Charts](#) 3.1+ deployed on that cluster. Please see [here](#).

If you need an ingress controller, you can install the [NGINX controller](#). Most of the modern Cloud vendors (Amazon Web Services, Microsoft Azure, Google Cloud Platform) already give you an ingress controller.

You should also have received credentials from your GENESYS representative to download container images. If this is not the case, please contact us.

Installing Universal Messaging

The automated way

The easiest way to install Universal Messaging is the automated way. It takes care of creating the Kubernetes cluster and all that is needed to run the application.

You can find the install script [there](https://artifacts.genesyslab.com/universal-messaging-install.sh): <https://artifacts.genesyslab.com/universal-messaging-install.sh>

On the plus side, you just have to provide coffee... On the negative side, you have less control over the way the cluster is built. If you prefer creating the cluster yourself, please go directly to the next paragraph.

Here is the simplest way to deploy Universal Messaging, say on Microsoft Azure (the options here are mandatory):

```
./install.sh --flavor aks \  
--release prod \  
--registry-username your-genesys-username \  
--registry-password your-genesys-password \  
--registry-email your-email
```

You will see some warning about generating passwords for the application, Redis Database, and RabbitMQ. You can provide each of them in the options.

Supported flavors are:

- `aks` , [Microsoft Azure](#)
- `eks` , [Amazon Web Services](#)
- `gke` , [Google Cloud](#)
- `microk8s` , [Micro K8S](#)

Here are all the options you can use (mandatory options are marked with a *):

- `--api-root fqdn-url`
The URL used to reach the Universal Messaging API.

- `--api-password password`
The password for the API admin user, will be stored in a Kubernetes Secret,
- `--cluster name`
The name of the Kubernetes Cluster to be created (when it applies),
- `--config path` (default: `./config.json`)
The filename used to store the generated Helm configuration,
- `--dry-run`
Commands are not executed, use this to see what would be executed,
- `--flavor name *`
The Kubernetes flavor to prepare.
Values: aks, eks, gke, microk8s,
- `--helm-debug`
Will run Helm in debug mode, providing more information,
- `--namespace name` (default: `messaging`)
Universal Messaging will be deployed in this Kubernetes namespace,
- `--rabbitmq-password password`
The password for the RabbitMQ admin user, will be stored in a Kubernetes Secret,
- `--rabbitmq-erlang-cookie cookie`
The Erlang Cookie for RabbitMQ, will be stored in a Kubernetes Secret,
- `--redis-password password`
The password for the REDIS database, will be stored in a Kubernetes Secret,
- `--registry-username name *`
The username to use to download Container images from Genesys Container Registry,
- `--registry-password password *`
The password to use to download Container images from Genesys Container Registry,
- `--registry-email email *`
The email to use to download Container images from Genesys Container Registry,
- `--release name *`
The name of the Helm Release to create,
- `--stage name`
Will install another Helm chart, docker images than the default production ones.
Try this only in your lab... Some stages are: `dev` , `beta` , `stage`
Other can exist to address various scenarii.
Stages can also alter the generated configuration,
- `--verbose` (default)
Will run verbosely, displaying more information,
- `--workers number`
The number of worker nodes in the Kubernetes cluster.
Some Kubernetes flavors have restrictions!

Some flavors can have additional options:

- Amazon Web Services `EKS`
- Microsoft Azure `AKS`
 - `--az-resource-group name *`
The Azure Resource Group to create,
 - `--az-resource-location name`
The Azure Resource Location to use.
This is **mandatory** if your account does not have a default location defined,
 - `--az-sunscription id_or_name`
The Azure Subscription to use for billing.

Notes:

- The API password **MUST** be complex (must have a score of at least 3 on `zxcvbn`, test site: <https://lowe.github.io/tryzxcvbn>), if not the `xx-set-admin-password job` will fail and the Helm Chart deploying will also fail.
- When installing on `microk8s` , you must be logged in the host machine that will run it.
- There is only 1 node possible with `microk8s` as of today. You can add more, but by yourself.
- When installing on `Azure` , the script will prevent the number of worker nodes to be less than 3.

The manual way

Before installing and running anything, we must add the [Genesys Helm repository](#):

```
helm repo add genesys https://charts.genesyslab.com
```

You can look at the charts that will be deployed by pulling them in advance:

```
helm pull genesys/universal-messaging
```

For reference, you can also pull the chart at a specific version, if you need to analyze what it will do:

```
helm pull https://charts.genesyslab.com/charts/universal-messaging-1.0.3.tgz
```

Then, we need to create a new Kubernetes namespace:

```
kubectl create namespace messaging
```

Since the container images for Universal Messaging are not public, we create the Kubernetes secret that will allow us to download them from the Genesys Docker registry:

```
kubectl create secret docker-registry \
  --namespace messaging regcred-genesys \
  --docker-server=cr.genesyslab.com \
  --docker-username=<your-name> \
  --docker-password=<your-password> \
  --docker-email=<your-email>
```

The Universal Messaging Application uses passwords to connect services to their RabbitMQ and Redis components. They are stored in [Kubernetes secrets](#). While, you can let [Helm](#) create them randomly, it is advisable to set them yourself. You get better control and you don't take the risk of losing access to Universal Messaging when you upgrade to a more recent version.

The API service administrator password is **required** during the install. Helm will complain if it is not provided. That password is not stored in a secret but in the API service database. It also must be complex otherwise the Helm installation will fail.

```
tee myconfig.yaml &>/dev/null <<EOM
global:
  redis:
    password: r3d1sS3cr3t
rabbitmq:
  auth:
    password: r2bb1tS3cr3t
    erlangCookie: MyV3ryB1gS3cr3tC00k1e
api:
  admin:
    password: s1ms3cr3t
  api_root: https://www.acme.org
EOM
```

Finally, we deploy Universal Messaging via Helm:

```
helm install \
  --namespace messaging \
  --values myconfig.yaml \
  genesys/universal-messaging
```

You can observe the deployment of all pods by running:

```
watch -n 1 kubectl get pods --namespace messaging
```

This chart will install [Redis](#) and [RabbitMQ](#) along with all services needed to run the Universal Messaging platform.

You should see all pods getting to the `running` state within a few minutes.

Notes:

- To configure the ingress, please read the section [Configuring HTTPS](#)
- If the Ingress Controller used by your platform requires an ingress class, you should add it in your config.yaml (here with `nginx` as an example):

```
ingress:
  className: "nginx"
```

- As keeping password in plain text is a dangerous thing to do, we advise you to use file encryption technologies such as [Terraform Vault](#), [Ansible Vault](#), [Mozilla SOPS](#).
- You can also let Helm create the passwords by itself (except for the API's admin password that is required). In that case, do not forget to backup their values before upgrading the Helm chart (See [Upgrading Universal Messaging](#)).
- The value for `api_root` is optional. See Chapter [Using the REST API](#) for a more detailed explanation.

Installing the beta/Development versions

By default, Helm will install the stable version of Universal Messaging and its services.

If you want to try a more recent version than the default (to test a new feature or a fix), you just need to modify which images are downloaded. Of course, Beta versions are less stable, and Development (dev) versions are very unstable.

We suggest to not do this in production, but on a test cluster, this can be useful.

Let's modify the yaml configuration we prepared previously (or better yet a copy of it):

```
tee -a myconfig.yaml >/dev/null <<EOM
  image: {tag: dev, pullPolicy: Always}
config: { image: {tag: dev, pullPolicy: Always} }
commander: { image: {tag: dev, pullPolicy: Always} }
amb-connector: { image: {tag: dev, pullPolicy: Always} }
bedore-connector: { image: {tag: dev, pullPolicy: Always} }
bizmsg-connector: { image: {tag: dev, pullPolicy: Always} }
disqus-connector: { image: {tag: dev, pullPolicy: Always} }
gbm-connector: { image: {tag: dev, pullPolicy: Always} }
infobank-connector: { image: {tag: dev, pullPolicy: Always} }
line-connector: { image: {tag: dev, pullPolicy: Always} }
media4u-connector: { image: {tag: dev, pullPolicy: Always} }
plussmessage-connector: { image: {tag: dev, pullPolicy: Always} }
telegram-connector: { image: {tag: dev, pullPolicy: Always} }
viber-connector: { image: {tag: dev, pullPolicy: Always} }
wechat-connector: { image: {tag: dev, pullPolicy: Always} }
gcloudcx-connector: { image: {tag: dev, pullPolicy: Always} }
pureconnect-connector: { image: {tag: dev, pullPolicy: Always} }
aws-provider: { image: {tag: dev, pullPolicy: Always} }
azure-provider: { image: {tag: dev, pullPolicy: Always} }
box-provider: { image: {tag: dev, pullPolicy: Always} }
dropbox-provider: { image: {tag: dev, pullPolicy: Always} }
google-provider: { image: {tag: dev, pullPolicy: Always} }
EOM
```

Upgrading Universal Messaging

To be able to upgrade the Universal Messaging deployment safely, you need to ensure the passwords, etc will not be overwritten by the upgrade process. This is particularly true in an [Helm](#) environment such as the one we use: `helm upgrade`

Before upgrading, if a YAML configuration file was not created, all passwords must be saved as they would get reset during the upgrade process:

```
cat <<EOM > config.yaml
global:
  redis:
    password: $(kubectl get secrets -n messaging \
      -l app.kubernetes.io/name=api-redis \
      -o jsonpath="{.items[0].data.redis-password}"|base64 --decode)
rabbitmq:
  auth:
    password: $(kubectl get secrets -n messaging \
      -l app=rabbitmq \
```

```

    -o jsonpath="{.items[0].data.rabbitmq-password}"|base64 --decode)
  erlangCookie: $(kubectl get secrets -n messaging \
    -l app=rabbitmq \
    -o jsonpath="{.items[0].data.rabbitmq-erlang-cookie}"|base64 --decode)
EOM

```

You might also need to add some fields that you used during the installation.

We should also backup the Redis Database:

```

gum-cli backup --host host --password xxx | \
gzip >| backup-$(date +%Y%m%d%H%M%S).json.gz

```

Once this is done we can launch the upgrade:

```

helm upgrade rrr -f config.yaml genesys/universalmessaging

```

Where `rrr` is the Helm Release name and replace `config.yaml` with the YAML file where you keep the configuration.

Like before, you can observe the deployment of all pods by running:

```

watch -n 1 kubectl get pods --namespace messaging

```

Fine tuning

Deploying only the services you need

It is possible to not deploy some of the services, if you know you will never need them in your Universal Messaging instance.

To do so, mark their `enabled` property to `false` in your Helm YAML configuration, like this:

```

bedore-connector:
  enabled: false

```

```

dropbox-provider:
  enabled: false

```

Extra logging

When you are in the testing phase, you can also turn up the logging to get more information in the logs about what is happening.

To do so, you can simply modify the `logging` property in your Helm YAML configuration:

```

line-connector:
  logging:
    level: DEBUG

```

Logs are also flushed every so often or when an error occurs (written to their output stream, typically stdout). If you need to ensure the logs are written when they are generated, change their flush frequency:

```

line-connector:
  logging:
    flushFrequency: immediate
    level: DEBUG

```

You can also set default log setting at the global level that are superseded locally:

```

global:
  logging:
    level: DEBUG

line-connector:
  logging:
    level: TRACE

```

Here all services will log at `DEBUG`, instead of `INFO`, and `line-connector` will log `TRACE`.

Log levels are (in decreasing order):

- FATAL
- ERROR
- INFO
- DEBUG
- TRACE

The lower in this list the more verbose the logging becomes and the more impact it has on performance. The Default value is `INFO`.

Horizontal Auto-Scaling

By default, Horizontal Auto-Scaling is configured for the services as follows:

```
line-connector:
  autoscaling:
    enabled: true
    minReplicas: 1
    maxReplicas: 10
    targetCPUUtilizationPercentage: 80
```

You can change these values in your own YAML configuration.

Use a Custom Container Registry

The Container images for Genesys services are provided by <https://cr.genesyslab.com> at specific versions.

It is possible to use a custom registry to download them faster or simply to alleviate any downtime of the original Container Registry.

Most of the Cloud vendors offer a per-cluster registry. Just download the container images using [Docker](#) and upload them to your registry:

```
docker login --username xxx cr.genesyslab.com
docker pull cr.genesyslab.com/gum/line_connector
docker tag cr.genesyslab.com/gum/line_connector asia.gcr.io/gum/line_connector
docker push asia.gcr.io/gum/line_connector
```

Then, change your Helm YAML configuration as follows:

```
line-connector:
  image:
    registry: asia.gcr.io
    repository: gum/line_connector
    tag: xxx
    pullPolicy: IfNotPresent
```

If the `tag` property is not given in your config.yaml, the Helm Chart will use the value from `.Chart.AppVersion`, which is the most common situation. Please note that using tags like `latest` is considered dangerous as it leads to inconsistencies since Kubernetes does not know when the container image has actually changed.

If you use `microk8s`, you will set the registry to `none` and the pull policy to: `Never`, and uploading to microk8s registry is done as follows:

```
docker pull...
docker save cr.genesyslab.com/gum/line_connector line_connector.tar
microk8s ctr image import line_connector.tar
```

Using external Redis or RabbitMQ

Genesys uses the [Redis Chart](#) and [RabbitMQ Chart](#) from [Bitnami](#) when deploying Universal Messaging, you can configure them as you wish in your YAML configuration.

If you prefer to use external Redis or RabbitMQ to host the databases or process the message queuing, you just need to disable them and configure the access in your YAML configuration.

For example, you might want to use the Universal Messaging platform with an external RabbitMQ Messaging, such as [Amazon MQ](#) or [CloudAMQP](#).

You would simply disable the deployment of RabbitMQ and use the RabbitMQ FQDN from your provider:

```
global:
  rabbitmq:
    host: xyz.rmq.cloudamqp.com
    vhost: "vhostFromCloudAMQP"
    user: "userFromCloudAMQP"
  rabbitmq:
    enabled: false # Turn off local deployment of Bitnami's RabbitMQ
```

And create the Kubernetes secret that will hold the password:

```
kubectl create secret generic rrr-rabbitmq \
  --namespace messaging \
  --from-literal rabbitmq-password="p@sswordFr0mCl0ud@mqp!"
  --from-literal rabbitmq-erlang-cookie="ErlangCooki000123From12344CloudAQMP"
```

where `rrr` is the Helm Release name

Resource limits and requests

All Helm Charts are configured with Kubernetes resource limits and requests. You can change them as you see fit in your YAML configuration, as follows (these numbers are just an example):

```
line-connector:
  resources:
    limits:
      cpu: 500m
      memory: 512Mi
    requests:
      cpu: 200m
      memory: 128Mi
```

You can find the default values in the Helm Chart of each service. The connector services use these values:

```
resources:
  limits:
    cpu: 50m
    memory: 64Mi
  requests:
    cpu: 20m
    memory: 32Mi
```

The storage providers use these values:

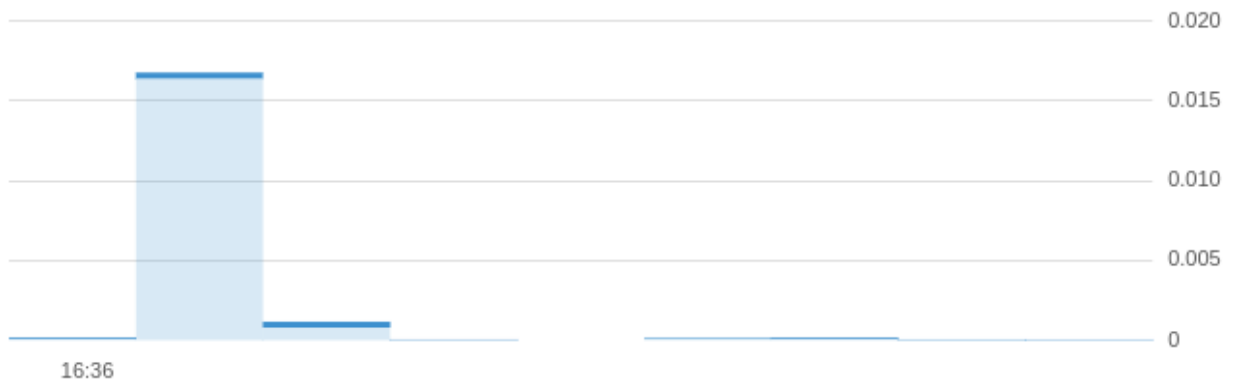
```
resources:
  limits:
    cpu: 100m
    memory: 64Mi
  requests:
    cpu: 20m
    memory: 32Mi
```

These values were calculated by load testing messages to a Kubernetes instance. The test consisted of blasting 10,000 text messages to the Social Messages over a few seconds and 1,000 1-MiB images and monitoring the CPU and memory used by the services.

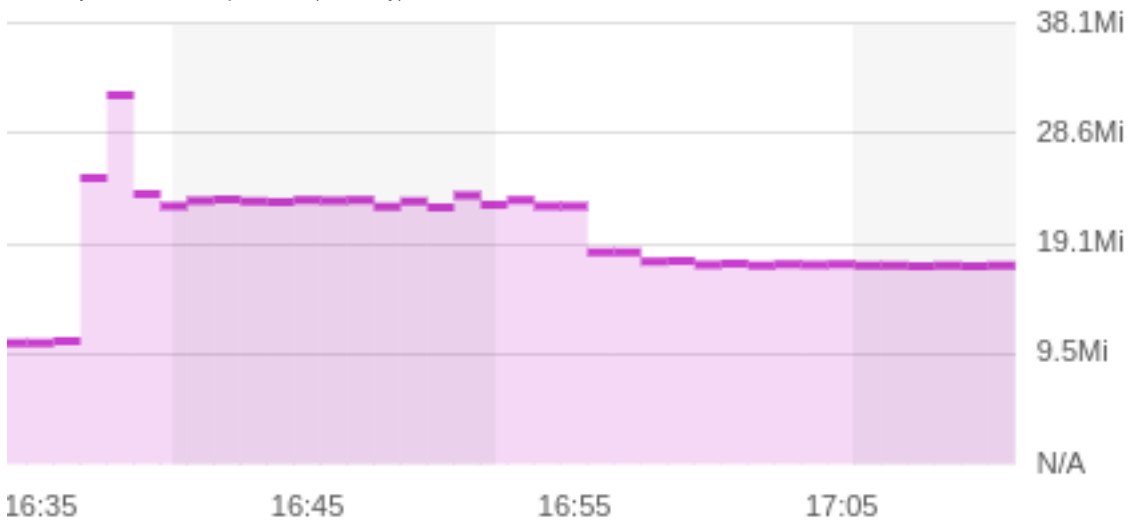
Here is the memory usage (in MiB) of the LINE service, during the image test:



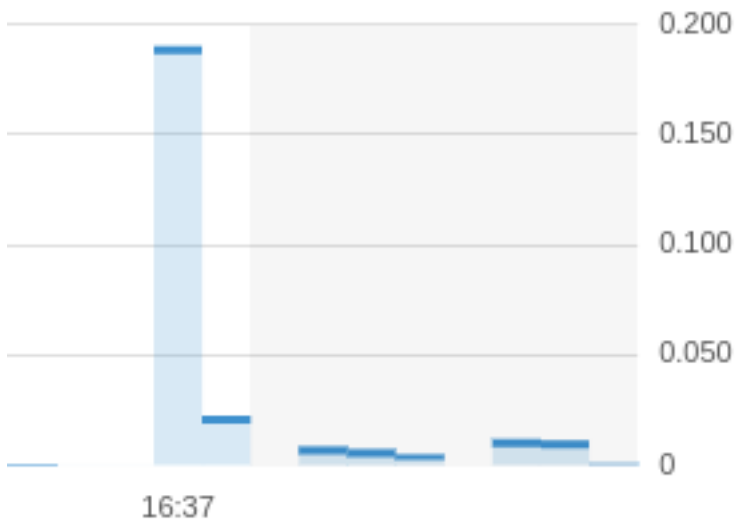
And the CPU usage:



Similarly, for the Azure provider (memory):



And CPU usage:



Note: The services are written in Go language. Go has a particular way of handling memory allocation. It does not release the memory released by applications quickly. This allows Go applications to allocate memory back very quickly in case another need comes soon. That explains why we do not see the memory going down rapidly on the memory graphs.

Configuring HTTPS

In order to connect Universal Messaging to Social Messaging services, you will need to configure outside accesses to the Social Messaging connectors.

The communication is ensured via https. Which means you will need a Fully Qualified Domain Name (FQDN).

Depending on your deployment and your existing infrastructure, this can be achieved in different ways.

1. You already have an external DNS on the Internet and can set an FQDN there,
2. You do not have that or do not want to, you will need to ask the Cloud provider to create an FQDN for you.

The first case is fairly straight forward as it will require only some DNS editing once you get the static Internet IP address. Also, if the cloud is your own Datacenter (Virtual Machines, Private Cloud, Bare Metal, etc), this will be, most probably, your only solution.

The second case depends heavily on the cloud you use and can be set up only after the LINE integration is already installed (as we need some services installed). We will describe the process for a few clouds in a moment.

Getting an FQDN with AWS

You first need to own an IP Domain. If you do not have any, the easiest is to get one from Amazon's Route 53 service.

To do that, from the AWS Console go to the Route53 service, click on "Register Domain" and follow the instructions.

Then go to the "Certificate Manager" service, click on "Request a Certificate", request a public certificate and add your domain "*.mydomain.com".

Once you get the domain and the certificate, you can get the arn of the latter and update the Kubernetes Ingress annotations:

```
metadata:
  annotations:
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTPS":443}, {"HTTP":80}]'
    alb.ingress.kubernetes.io/certificate-arn: arn:aws:acm:ap-northeast-1:123456:certificate/xxxx
    alb.ingress.kubernetes.io/actions.ssl-redirect: '{"Type": "redirect",
      "RedirectConfig": { "Protocol": "HTTPS", "Port": "443", "StatusCode": "HTTP_301"}}'
```

Make sure the third line fits in one line, we had to split it in two to fit on the document page.

Use the arn of the certificate from the AWS console.

Finally add a new CNAME to your DNS zone and alias it to the ALB DNS from the ingress. This can be done directly in the AWS Console or with the command line:


```

target=$(kubectl get ingress \
  --namespace messaging \
  prod-messaging \
  --output jsonpath='{.status.loadBalancer.ingress[0].hostname}' \
)
cat > cname.json <<EOM
{
  "Changes": [{
    "Action": "CREATE",
    "ResourceRecordSet": {
      "Name": "messaging.acme.com",
      "Type": "CNAME",
      "AliasTarget": {
        "DNSName": "${target}",
        "EvaluateTargetHealth": false
      }
    }
  }
]
}
EOM
zoneid=$(aws route53 list-hosted-zones --output json \
  jq -r '.HostedZones[] | select(.Name == "acme.com.") | .Id' |\
  cut -d/ -f3
)
aws route53 change-resource-record-sets \
  --hosted-zone-id $zoneid \
  --change-batch file://cname.json

```

Note: static IP addresses, names, and certificates are paying options with Amazon Web Services and they are not the responsibility of Genesys.

Getting an FQDN with Azure

First you need to get the `helm` release name and the external IP address that was assigned to the `ingress` service (reverse proxy):

```

RELEASE=$(helm list --namespace messaging --output json | jq -r '.[0].name')
PIP=$(kubectl get ingress \
  --namespace messaging \
  --selector "app.kubernetes.io/instance=$RELEASE" \
  --output jsonpath='{.items[0].status.loadBalancer.ingress[0].ip}')

```

Then you need to ask Azure for an FQDN:

```

RESOURCEGROUP=$(az network public-ip list
  --query "[?ipAddress!=null]|[?contains(ipAddress, '$PIP')].resourceGroup]"
  --output tsv)
PIPNAME=$(az network public-ip list \
  --query "[?ipAddress!=null]|[?contains(ipAddress, '$PIP')].name]" --output tsv)
az network public-ip update \
  --resource-group $RESOURCEGROUP \
  --name $PIPNAME \
  --output table \
  --dns-name acme-messaging

```

Replace the `dns-name` option with the name you want to use. After a few minutes you should get a response from `aks` in the form of a JSON object.

The property `.dnsSettings.fqdn` will contain the fqdn you can use for your LIS installation.

The next step is to get a certificate that will authenticate this FQDN. This will depend on the certificate provider you use.

If you forget the FQDN, you can always get it from Azure:

```

az network public-ip list \
  --query "[?ipAddress!=null]|[?contains(ipAddress, '$PIP')].dnsSettings.fqdn" \

```

```
--output tsv
```

Note: static IP addresses and certificates are paying options with Microsoft Azure and they are not the responsibility of Genesys.

Getting an FQDN with Google Cloud Platform

First, you need to get a static IP address for your Kubernetes cluster:

```
gcloud compute addresses create my-static-ip --global
```

Once this is done, you can get its value like this:

```
gcloud compute addresses describe my-static-ip --global
```

Update the Ingress definition with the following annotation:

```
kubernetes.io/ingress.global-static-ip-name: my-static-ip
```

Also, all services accessible via the ingress must have the type `NodePort`, so you should update them in your config.yaml. For example:

```
api:
  service:
    type: NodePort
```

Note: It can take a few minutes for the Ingress to reflect the IP address.

For the FQDN, you have several choices:

- you can ask Google for an FQDN at [Google Domains](#)
- you have your own FQDN, you just need to point your nameserver to the static from Google, and configure your SSL certificate accordingly.
- You can also use [nip.io](#) or [sslip.io](#) to get an FQDN automatically (typically, the FQDN is `<static-ip>.nip.io`)

You can also use a [Google-managed SSL certificate](#).

Note: static IP addresses, names, and certificates are paying options with Google and they are not the responsibility of Genesys.

Logging Configuration

We recommend you to use the logging solution that is available with your cloud provider.

- On [Google Cloud](#), the best is to use the default [Stackdriver logging](#).
- On [Microsoft Azure](#), [Log Analytics](#) will be the preferred way.
- On [Amazon Web Services](#), using the [Amazon Elastic Container Service for Kubernetes \(EKS\)](#), the logs are integrated automatically with [Amazon CloudWatch](#) and [CloudTrail](#).

Now, if you want to run your own logging support, it is possible as well. There is a good tutorial on how to deploy [Elasticsearch](#) and [Kibana](#) on the [Kubernetes website](#) itself.

See also:

- [Setting Up Logging with Kubernetes](#)

You can also read the logs via [kubectl logs](#) directly:

```
kubectl logs -n messaging -1 --tail --follow -l connector=gcloudcx | \
  bunyan -L -o short
```

To pretty-read the logs, please get the [bunyan log reader](#) or run it through [bunyan in Docker](#)

Tips and Tricks for Checking the deployment

Access the Redis Database

To connect to the Redis database (from within the cluster), you can run a container with the Redis Client (`XXX` is the name of the Helm Release):

```
REDIS_PASSWORD=$(kubectl get secret \
  --namespace xyz RRR-redis -o jsonpath="{.data.redis-password}" \
  | base64 --decode)
kubectl run --namespace xyz redis-client --rm -it \
  --image bitnami/redis:6.2.5 -- redis-cli \
  -h RRR-MMM-redis-master-0.xyz.svc.cluster.local \
  -a $REDIS_PASSWORD
```

Replace `xyz` by the Kubernetes namespace, `RRR` by the Helm release name of your deployment, and `MMM` the microservice name ("gcloudcx-connector", for example).

Or, if you have `redis-cli` installed on your machine, you can forward ports with Kubernetes:

```
kubectl port-forward service/xyz-MMM-redis-master 6379:6379
redis-cli -a $REDIS_PASSWORD
```

Then, you can use the REDIS query language to check things, e.g.:

```
keys config:*
```

Backing up the Redis Database

Genesys provides a client tool to backup and restore the database.

You can download it from:

- Linux: <https://artifacts.genesyslab.com/gum-cli-1.0.12.linux.7z>
- MacOS: <https://artifacts.genesyslab.com/gum-cli-1.0.12.macos.7z>
- Windows: <https://artifacts.genesyslab.com/gum-cli-1.0.12.windows.7z>
- Docker: cr.genesyslab.com/gum/gum-cli

To backup the database, you just run the following:

```
gum-cli backup --host host --password xxx --output backup.json
```

Where `xxx` is the administrator password for the API.

You can run some more complex command that will compress the backup and use the current date in the filename:

```
gum-cli backup --host host --password xxx | \
  gzip >| backup-$(date +%Y%m%d%H%M%S).json.gz
```

To restore the database, simply run:

```
gum-cli restore --host host --password xxx -i config.json
```

If the backup was compressed, then you would run this:

```
zcat backup-20210805230406.json.gz | gum-cli restore --host host --password xxx
```

You can store the default values for `gum-cli` in `$HOME/.config/gum-cli/config` or `$HOME/.gum-cli` in YAML:

```
host: host.local.net
password: xxx
```

Then you do not need them on the command line anymore: `gum-cli backup -o config.json`

You can get logs from `gum-cli` execution by doing thing:

```
gum-cli backup --log mylog.log --host host --password xxx --output backup.json
```

To get more verbose logs:

```
DEBUG=1 gum-cli backup --log mylog.log --host host --password xxx --output backup.json
```

To read the logs, please get the [bunyan log reader](#) or run it through [bunyan in Docker](#)

Finally, you can get the documentation by running:

```
gum-cli --help
```

Access the RabbitMQ Dashboard

If you did not set your own user, the chart will set it to `user` .

if you did not set your own administrator password, the chart will create one. To retrieve the password:

```
RABBITMQ_PASSWORD=$(kubectl get secret \
  --namespace xyz RRR-rabbitmq -o jsonpath="{.data.rabbitmq-password}" \
  | base64 --decode)
```

The RabbitMQ service is accessible to the pods in the cluster on port `5672` and its web dashboard on port `15672` at the DNS name: `RRR-rabbitmq.xyz.svc.cluster.local`

Then forward the dashboard port of RabbitMQ:

```
kubectl port-forward --namespace xyz service/RRR-rabbitmq 15672:15672
```

Finally, go to `[http://127.0.0.1:15672]` to get to the RabbitMQ Management site.

Replace `xyz` by the Kubernetes namespace and `RRR` by the Helm release name of your deployment.

Chapter 3

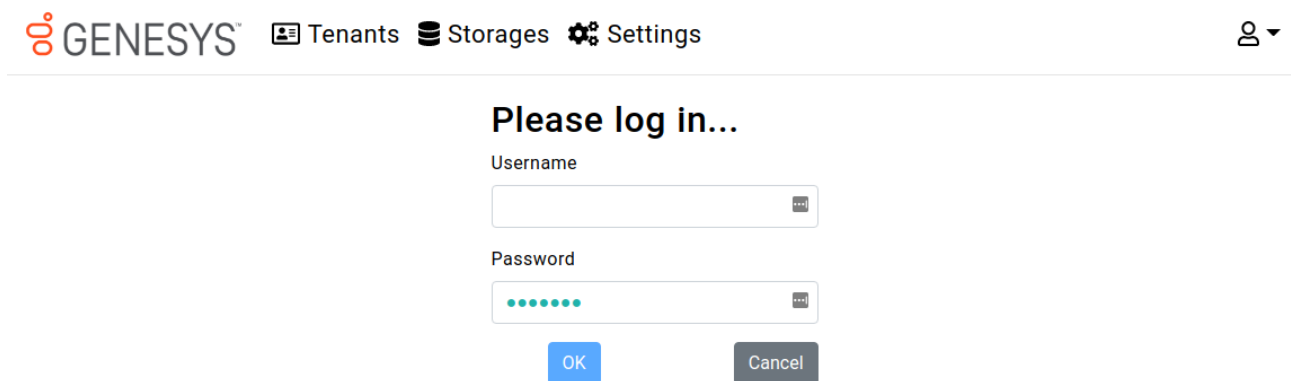
Configuration

The Universal Messaging application can be configured either via a website or its own REST API. The latter will be explained in the next chapter.

The configuration website is available at the main URL of the application. For example:

`https://messaging.acme.com`

When you get connected, you need to enter the administrator's credentials that were configured during the installation:



The screenshot shows the Genesys configuration website interface. At the top left is the Genesys logo. To its right are navigation links: 'Tenants' (with a list icon), 'Storages' (with a database icon), and 'Settings' (with a gear icon). On the far right is a user profile icon with a dropdown arrow. Below the navigation is a login form titled 'Please log in...'. It contains two input fields: 'Username' and 'Password'. The 'Password' field is masked with green dots. Below the fields are two buttons: 'OK' (blue) and 'Cancel' (grey).

Once you are logged in, you can configure the Universal Messaging application.

Note: You can change the language of the website with the Language Menu in the bottom-right.

Common Settings

In the "Settings" page, you can configure things such as:

- the Outbound Proxy. All services that access the Internet will use that value.
- the Notifier Destinations. Allows services to "tell" when they start/stop, ...
- the Default Commander (See the section about Commanders)

Notifier Destinations

You can set any number of supported Notifier Destinations. Services will send messages to all of them.

Typically, services will send messages when after they have started successfully and before they shutdown. If a service uses more than one Kubernetes Pod, every Pod will send notifications.

Note: Setting Notifier Destinations, while very useful, does **not** replace proper monitoring and alerting that you should set up at the Kubernetes level (AWS Cloudwatch, Azure Monitor, Google Cloud Monitoring, etc). When Pods cannot start *enough*, they will never send notifications, and supporting Kubernetes Deployments like Redis and RabbitMQ, being third party applications, cannot send notifications. For example, if the Redis database does not start, the connector services that uses it will also not start properly, but since it needs the database to retrieve its configuration, it will not be able to send notifications.

Slack

To setup [Slack](#), you need to create an account on their website (you can use your Google Account or your AppleID, or any email account).

Once you have created an account, click on the "CREATE A NEW WORKSPACE" button on the top right of the page. You can also do this from the [Slack](#) application on your Desktop or your smartphone.

Type the name of the workspace you want to create:

Step 1 of 3

What's the name of your company or team?

This will be the name of your Slack workspace – choose something that your team will recognize.

 38

Let anyone with an @genesys.com email join this workspace.

Next

Add a new channel:

Step 2 of 3

What's your team working on right now?

This could be anything: a project, campaign, event, or the deal you're trying to close.

 70

Next

Step 3 of 3

Who do you email most about **production**?

To give Slack a spin, add a few coworkers you talk with regularly.

[+](#) Add another

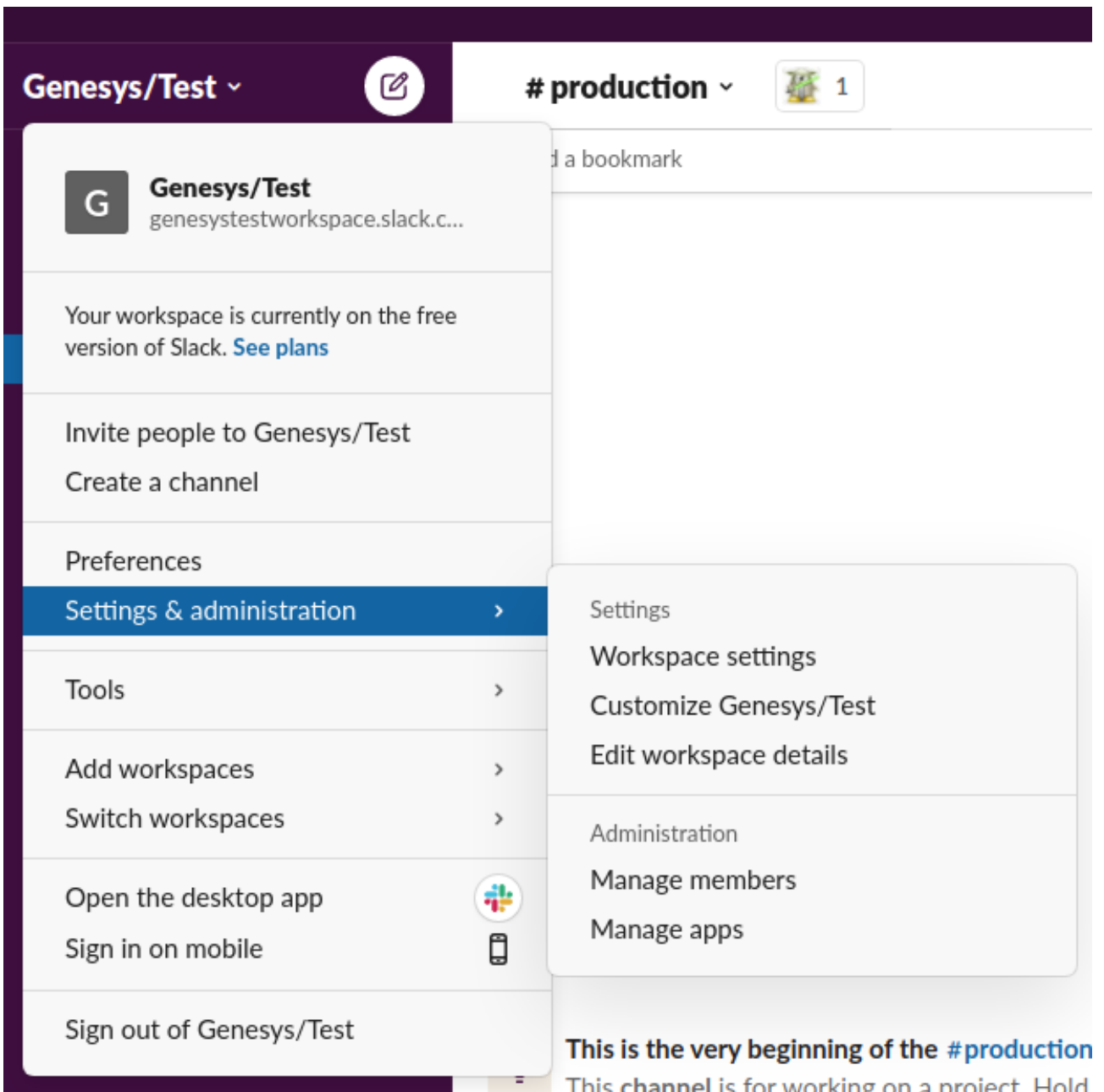
[↗](#) Get a sh

Add Teammates

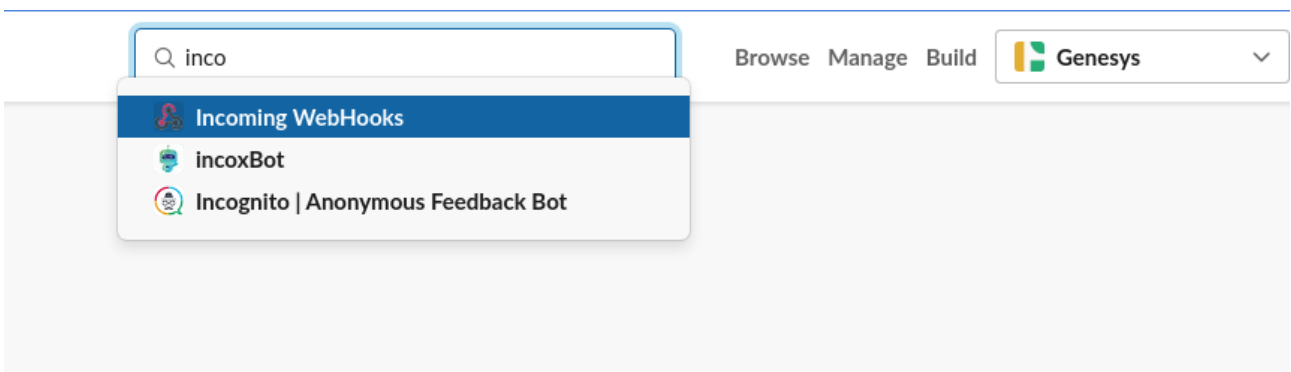
Skip this step

You can add team members or skip the next step:

After the workspace is started, go to its settings:



Click on "Configure Apps", and start typing "Incoming Webhooks" in the "🔍 Search App Directory" input on the top right of that page:



And click on the "Add to Slack" button.

Once added, select the channel that will receive notifications, and click on "Add Incoming Webhooks integration":

Post to Channel

Start by choosing a channel where your Incoming Webhook will post messages to.

production ▼

[or create a new channel](#)

Add Incoming WebHooks integration

By creating an incoming webhook, you agree to the [Slack API Terms of Service](#).

Once added, copy the Webhook URL from that page and paste it on the Universal Messaging Settings page:


Settings

Outbound Proxy 🔒

Proxy URL

`http://squid.local:3124`

Notifier Destinations

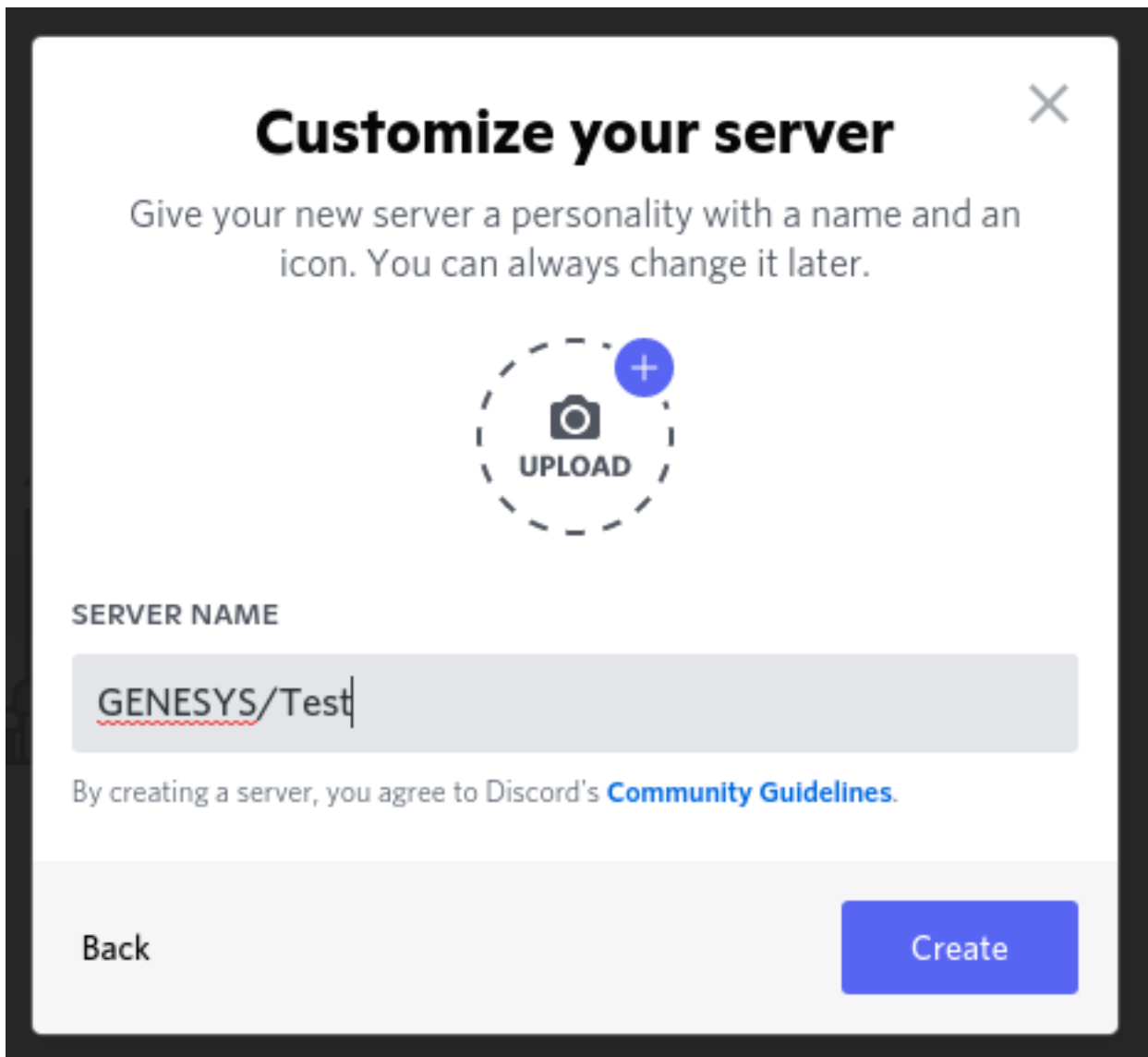
Type	Webhook URL	Channel
Slack ✓ ⌵ 	<code>https://hooks.slack.com/services/T02UNJ1TQ5A/B03A9TX5A1</code> ✓	#production ✓ +

And click on the "+" sign.

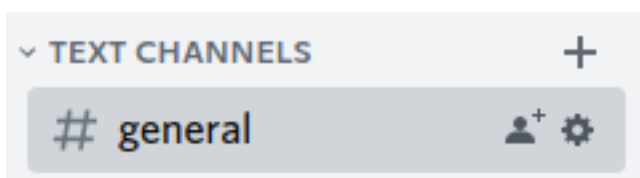
Discord

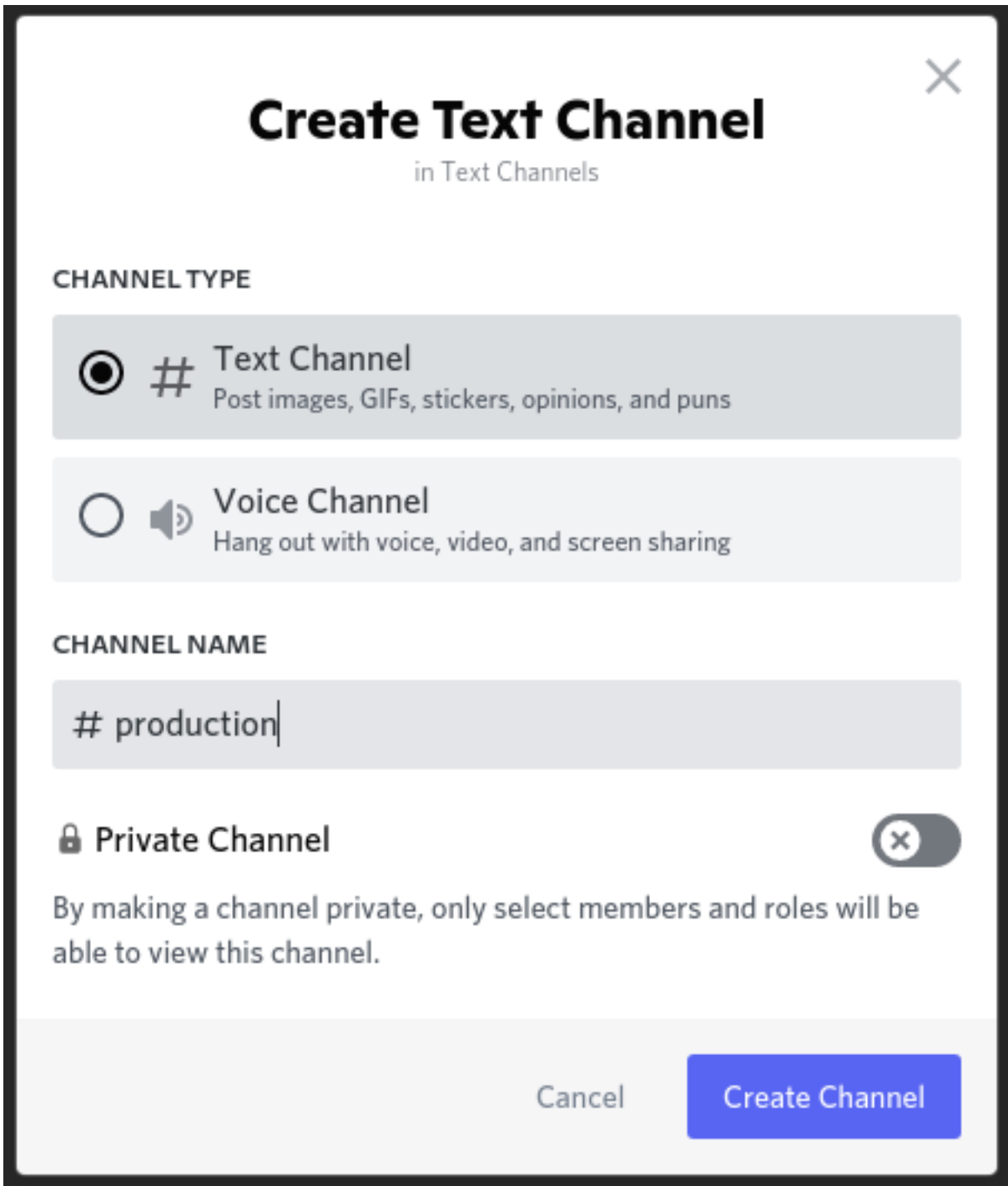
To setup [Discord](#), you need to create an account on their website.

Then, click on the "+" button on the left bar, to "Add Server" and follow the dialogs:

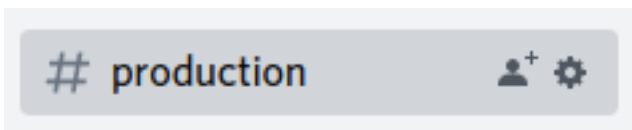


Once created, you can add new text channel, by click on the "+" sign in the "TEXT CHANNELS" collection:

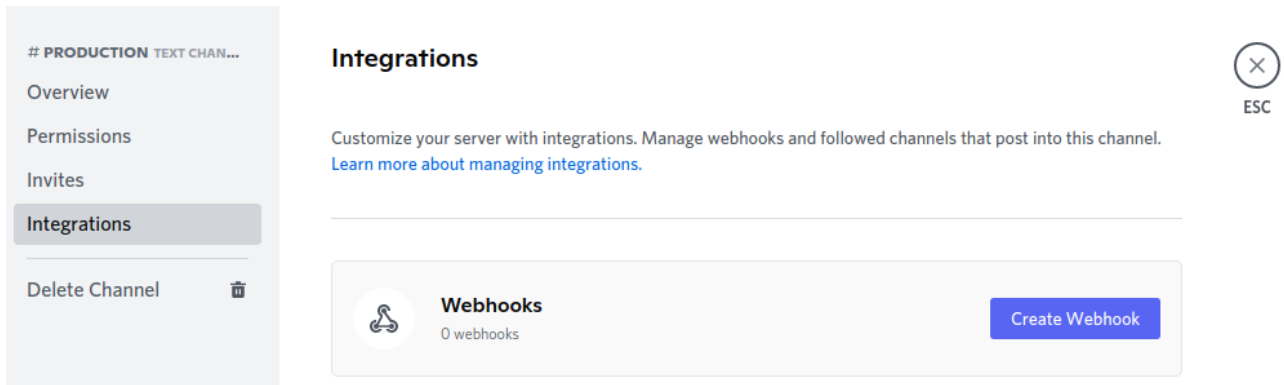




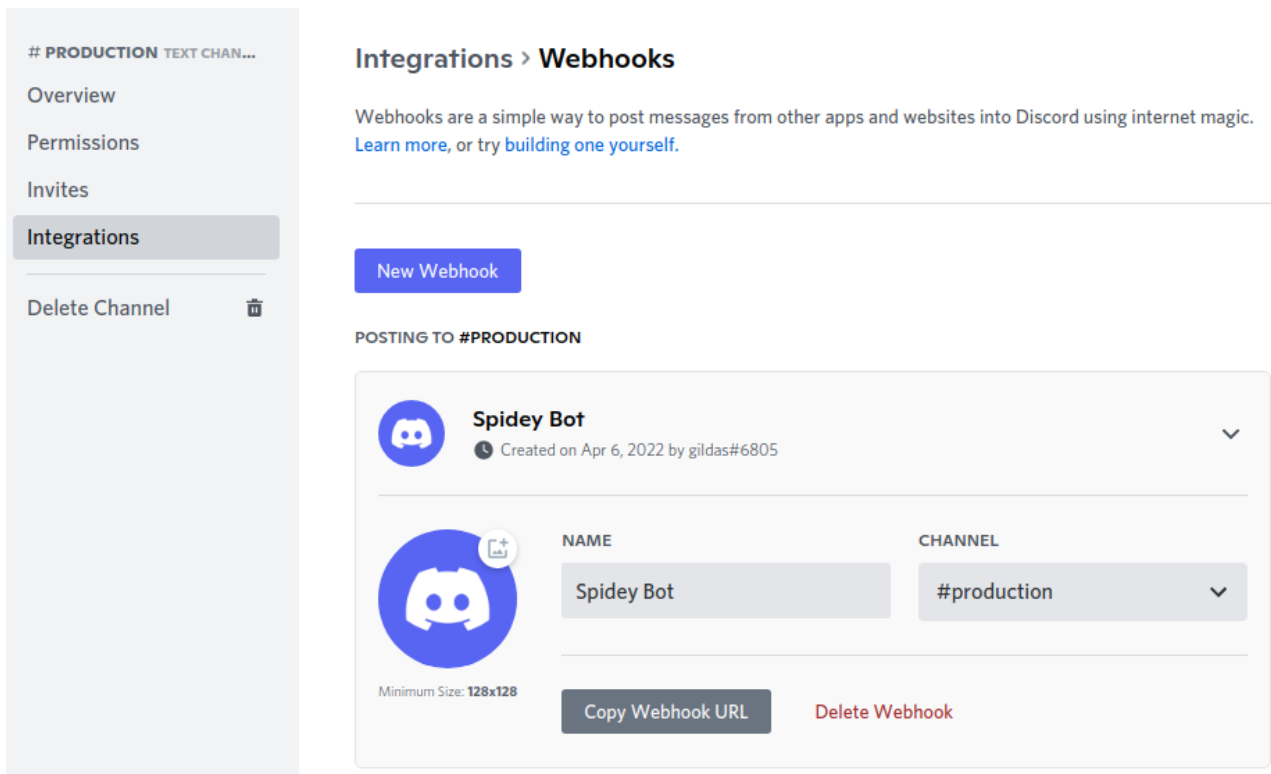
Once created, click on the icon, to edit the settings:



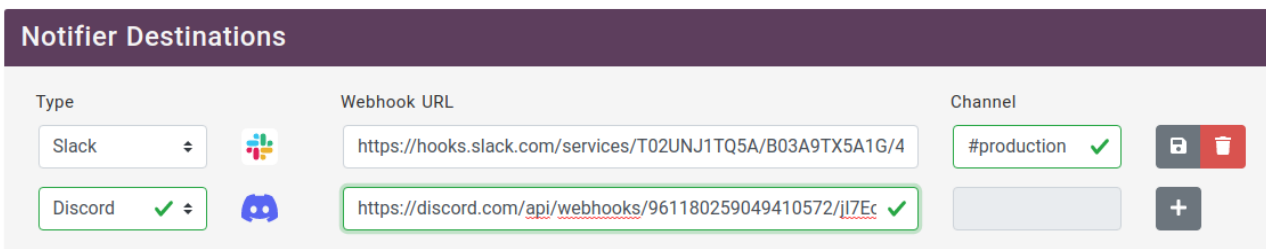
And select "Integrations":



Click on the "Create Webhook" button, and copy the Webhook URL:



Paste the URL on the Universal Messaging Settings page:



And click on the "+" sign.


Commanders

Commander is the service that can execute commands sent by the agents from the CX Platforms.

The most common command, as of today, is used to allow the agent to continue a conversation with their guest at a later moment.

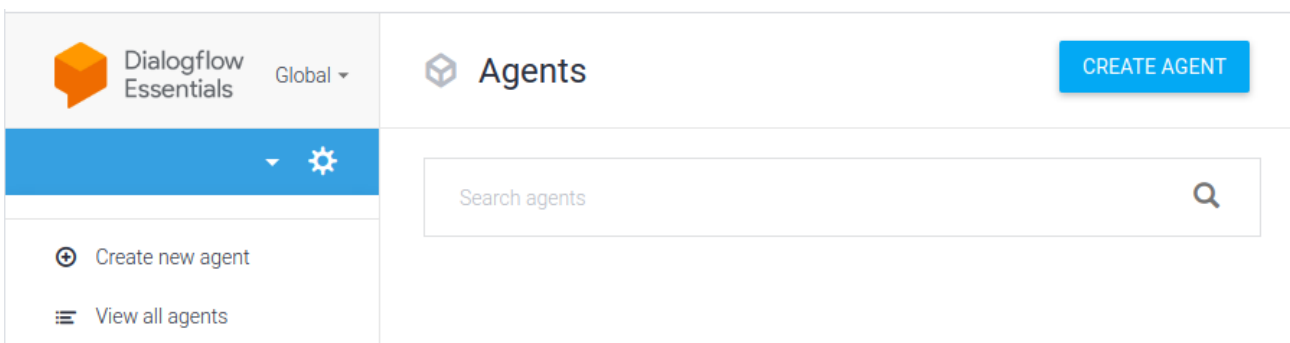
Commander configuration can be done on the Settings page, for the Default Commander, or in the Tenant's Commander tab.

The Commander configuration looks like this:

Google Dialogflow™	Dialogflow Agent Code?	<input type="text" value="dialogflow-command-continue-1.0.0.zip"/>
	Location	<input type="text" value="Select One..."/>
	Language	<input type="text" value="Select One..."/>
	Environment Id	<input type="text" value="Enter a Dialogflow Environment Identifier (e.g.: draft)"/>
Google Cloud Authentication	Project Identifier	<input type="text" value="Enter an identifier"/>
	Client Identifier	<input type="text" value="Enter an identifier"/>
	Client Email	<input type="text" value="Enter an email"/>
	Private Key Identifier	<input type="text" value="Enter an identifier"/>
	Private Key	<input type="text" value="Enter a private key"/>

The link to the zip file contains the code to import in Google Dialogflow. The location, language, Environment Id are given when you configure the agent in Google Dialogflow.

To configure Google Dialogflow, you first need to create an Agent on your Google Dialogflow console:



Click on "CREATE AGENT" and fill in the form:

continue CREATE ⋮

DEFAULT LANGUAGE ? **DEFAULT TIME ZONE**

English – en ▼ (GMT+9:00) Asia/Tokyo ▼

Primary language for your agent. Other languages can be added later. Date and time requests are resolved using this timezone if not provided in the API requests.

GOOGLE PROJECT

genesys-line-test ▼

Enables Cloud functions, Actions on Google and permissions management.

AGENT TYPE

Set as Mega Agent

Combine multiple Dialogflow agents (i.e. sub agents) into a single agent (i.e. [mega agent](#)).

Here, we gave the agent the name of the Commander command. The actual name does not matter much.

Once created, you need to go to the settings of the agent and click on the "IMPORT FROM ZIP":

continue

SAVE

General Languages ML Settings **Export and Import** Environments Speech Share

EXPORT AS ZIP

Create a backup of the agent

RESTORE FROM ZIP

Replace the current agent version with a new one. All the intents and entities in the older version will be deleted.

IMPORT FROM ZIP


Upload new intents and entities without deleting the current ones. Intents and entities with the same name will be replaced with the newer version.

The default Environment (from the same settings page) is called "draft". Its name needs to be filled in the Universal Messaging config webpage.

On the Google Dialogflow console, go to the agent's settings and click on the Project:

continue SAVE

General Languages ML Settings Export and Import Environments Speech Share



DESCRIPTION

Describe your agent (will be used in Web Demo integration)

DEFAULT TIME ZONE

(GMT+9:00) Asia/Tokyo

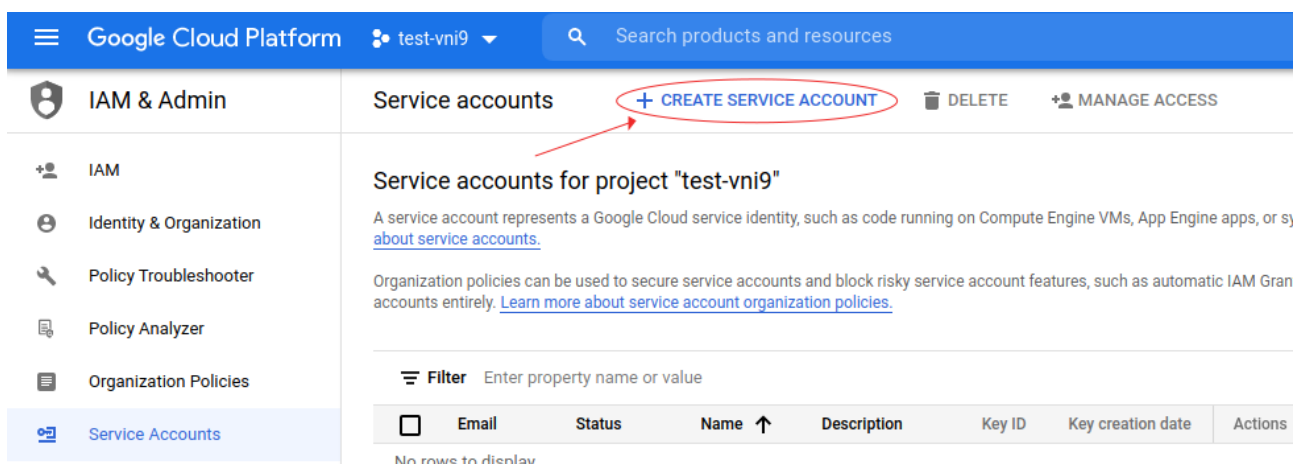
Date and time requests are resolved using this timezone if not provided in the API requests.

AGENT AVATAR URI
Define URI to agent avatar that will be used in [Web Demo](#) and [Hangouts Chat](#) integrations.

GOOGLE PROJECT

Project ID	<u>simcommander</u>
------------	---------------------

This will bring you to the Google Cloud Platform console. Go to the "IAM & Admin" section, and to "Service Accounts", and click on "CREATE SERVICE ACCOUNT":



Google Cloud Platform test-vni9 Search products and resources

IAM & Admin

- IAM
- Identity & Organization
- Policy Troubleshooter
- Policy Analyzer
- Organization Policies
- Service Accounts**

Service accounts + CREATE SERVICE ACCOUNT DELETE MANAGE ACCESS

Service accounts for project "test-vni9"

A service account represents a Google Cloud service identity, such as code running on Compute Engine VMs, App Engine apps, or [services](#). [Learn more about service accounts.](#)

Organization policies can be used to secure service accounts and block risky service account features, such as automatic IAM Gran accounts entirely. [Learn more about service account organization policies.](#)

Filter Enter property name or value

<input type="checkbox"/>	Email	Status	Name ↑	Description	Key ID	Key creation date	Actions
No rows to display							

And give the service account a name, then click on "DONE":

Create service account

1 Service account details

Service account name

simcommander

Display name for this service account

Service account ID

simcommander

@test-vni9.iam.gserviceaccount.com



Service account description

Describe what this service account will do

CREATE AND CONTINUE

2 Grant this service account access to project (optional)

3 Grant users access to this service account (optional)

DONE

CANCEL

Then go in the account setting by click on it on the service accounts list and go to the keys tab:

← simcommander

DETAILS PERMISSIONS **KEYS** METRICS LOGS

Keys

⚠ Service account keys could pose a security risk if compromised. We recommend you avoid downloading service account keys and instead use the [Workload Identity Federation](#). You can learn more about the best way to authenticate service accounts on Google Cloud [here](#).

Add a new key pair or upload a public key certificate from an existing key pair.

Block service account key creation using [organization policies](#).
[Learn more about setting organization policies for service accounts](#)

ADD KEY ▾

Type	Status	Key	Key creation date	Key expiration date
No rows to display				

Click on the "ADD KEY" and create a JSON key:

ADD KEY

Type

No rows to display

Create private key for "simcommander"

Downloads a file that contains the private key. Store the file securely because this key can't be recovered if lost.

Key type

JSON
Recommended

P12
For backward compatibility with code using the P12 format

CANCEL **CREATE**

The JSON authentication file is automatically saved on your computer. On the Universal Messaging Config page, you can either fill in each field of import the authentication JSON file using the small icon under "Google Cloud Authentication" on the config website.

Storages

Storages are used to store attachments from Social Media and/or CX Platforms when no public URL is available or restrictions within these systems preventing attachment access.

On the "Storage" tab, Just select the type of Storage you want to define and fill in its credentials from the provider.

Also, set the "Purge Delay" in seconds. This will used by the Storage Provider services to remove attachments that are not used anymore. After a chat is closed, the services will wait for the given purge delay and then delete the attachments related to that chat from their cloud storage.

You should select one of the storages you defined as the "Default Storage". That will be the one used by Tenant's Messaging and CX Platform by default.

Storage

⚠
+
↓
📁
🗑

Storage Type

Select One... ⌵

Storage Name

Enter a storage name

Default Storage

Purge Delay

300

seconds (~ 5 minutes)

Tenants

Tenants describe the relationship between a Social Media (Messaging) and a Customer Experience Platform.

The API Authentication allows IVR engines to send messages to Social Medias by sending REST Requests to the Universal Messaging API. The available Authentication types are:

- Basic
- None (which means no authentication necessary, not recommended)

Tenant

⚠
+
↓
📁
🗑

General
API Authentication
CX Platform
Commander
Messaging

Authentication Type

Basic ⌵

Username

Enter a username

Password

Enter a password

The CX Platform tab supports the following:

- Genesys Cloud CX,
- Genesys PureConnect

The Messaging tab supports the following:

- Apple Messages for Business,

- Bedore Web,
- BizM KakaoTalk,
- Disqus,
- Docomo +Messages,
- Google Business Messages,
- Infobank KakaoTalk,
- LINE,
- Media4U,
- Telegram,
- Viber,
- WeChat

CX

Genesys Cloud CX

On the third tab of the Tenant, you can choose the Customer Experience Platform (CX Platform). The first choice allows a connection with Genesys Cloud CX via its Open Messaging API.

Credentials

In the Genesys Cloud Administration console, go to "People & Permissions/Roles", and add a new Role, with the following permissions: `messaging:integration:all` , `conversation:message:receive` , `conversation:message:create` .

☰ People & Permissions / Roles / Permissions / OpenMessaging Integration

People

Roles / Permissions

Authorized Organizations

Divisions

Role Details
Permissions

Show: All Permissions Assigned Permissions

	Permission	Description	License	Conditions	Division Aware ?
<input checked="" type="checkbox"/>	Conversation > message > Create	Create a message	PureCloud 3		
<input checked="" type="checkbox"/>	Conversation > message > Receive	Receive a message	PureCloud 3		
<input checked="" type="checkbox"/>	Conversation > message > View	View conversation messages	PureCloud 3		
<input checked="" type="checkbox"/>	messaging > integration > All Permissions	Assigns all integration permissions, including any future permissions	PureCloud 3		
<input checked="" type="checkbox"/>	messaging > integration > Add	Create/Add an integration with a Messaging provider	PureCloud 3		
<input checked="" type="checkbox"/>	messaging > integration > Delete	Delete an integration with a Messaging provider	PureCloud 3		
<input checked="" type="checkbox"/>	messaging > integration > Edit	Update an integration with a Messaging provider	PureCloud 3		
<input checked="" type="checkbox"/>	messaging > integration > View	View integrations created with a Messaging provider	PureCloud 3		

Then, go to "Integrations/OAuth", and add a new Client and give it the new Role:

Integrations / OAuth / Universal Messaging Client

Integrations

Actions

Single Sign-on

OAuth

Authorized Applications

Client Details Roles

App Name

Universal Messaging Client

Description

This is the Client used by Genesys Universal Messaging

Token Duration (seconds): the number of seconds, between 5mins and 48hrs, until tokens created with this client expire.

86400

Grant Types

Client Credentials

Code Authorization

Token Implicit Grant (Browser)

SAML2 Bearer

Upon saving, you will receive a Client ID and a Client Secret. Note these as you will need them in Universal Messaging's Config.

Now open the Universal Messaging Config and add a new Tenant, then go to the "CX Platform" tab, and enter the values you received earlier:

Tenant

General API Authentication **CX Platform** Commander Messaging

Contact Type:

GENESYS Cloud CX

Associated Storage

Default Storage

Region

Japan

Client Identifier


4e56fb3-653f-4fbd-8fe0-dead1685beef

Secret

2354terdvfddh4wsd5t434tsdfbe45w3avbc

Open Messaging

Then, give your GCloudCX Open Messaging Integration a name, a token (You can choose any token you like, a UUID for example) and fill in the Integration WebHook URL:

Integration Name	Universal Messaging With XXX 
Integration ID	
Integration WebHook	https://there.acme.com/openmessaging 
Integration Token	sup3r s3cr3t 

Note: Make sure the Integration URL uses the endpoint you chose to receive all requests and ends with /openmessaging. Most often, if you used the standard Load Balancer given by your Cloud provider, the URL will be the address obtained when assigning the Load Balancer.

The Custom Data will be added to the Open Messaging's Participant Data as seen on the Agent's UI:

Some data is already set depending on the Messaging Connector that is configured with the Tenant:

- `media` ,
The name of the Messaging Connector, one of:
"Apple Messages for Business", "Bedore", "BizM KakaoTalk", "Disqus", "Google Business Messages", "Infobank KakaoTalk", "LINE", "Media4U", "Telegram", "Viber", "WeChat"
- `capabilities` ,
The comma-separated list of capabilities supported by the guest device as provided by the Social Media,
[Apple Messages for Business](#)
- `country` , The country where the guest is located,
Viber, WeChat
- `city` , The city where the guest is located,
WeChat
- `device` ,
The device type of the guest, Viber
- `forum` ,
The forum where the conversation was started,
Disqus
- `googleEntryPoint`
The Google Entry Point if present,
Google Business Messages
- `googlePlaceId`
The Google Place Identifier if present,
Google Business Messages
- `googleNearPlaceId`
The Google Near Place Identifier if present,
Google Business Messages
- `group`
The group identifier as provided by the Social Media.
[Apple Messages for Business](#)
- `intent` ,
The Intent that started this conversation (if any).
[Apple Messages for Business](#)

- `locale`
Apple Messages for Business, Google Business Messages, Viber, WeChat
- `province` , The province where the guest is located,
WeChat
- `subscribeScene` ,
The place where the guest is,
WeChat
- `thread` ,
The thread where the conversation was started,
Disqus

Agent Name

You can also choose the default name of the agent to show on the Social Media side (provided you choose "Prefix Agent Name" in the Messaging tab).

Commander usage

Check the "Use Commander" box, if this Tenant will use Commander. Uncheck "Default Commander", if you want to use a specific Commander configuration (see the Commander tab of that Tenant).


Enter a Commander Prefix that will be used by agents to send commands to the Commander:

Commander

Use Commander

Default Commander

Commander Prefix

Google maps

You can configure this section to display a nice map when the guests send their location to the agents, as show here:

This is the beginning of the conversation.



morning!

Feb 2, 2022 11:26 AM

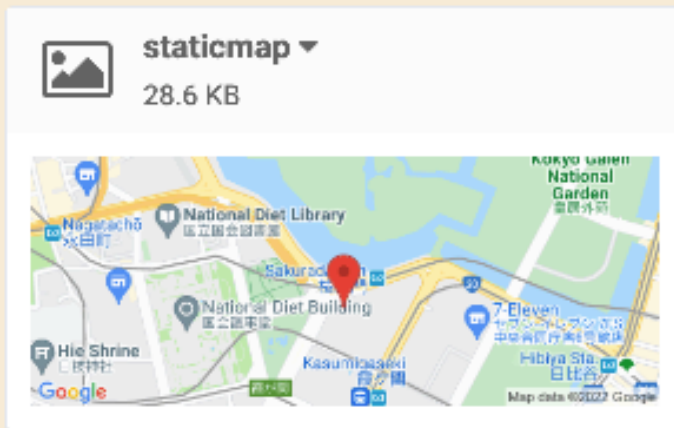
Welcome to GENESYS OpenMessaging



Feb 2, 2022 11:26 AM
Workflow



<https://www.google.com/maps/search/?api=1&query=35.676497%2c139.750250&hl=en>



Feb 2, 2022 11:29 AM

To compute the map, you need to enter your Google Maps Key and, eventually, Secret here:

Google Maps

Google Maps: These values can be set to display a preview of Locations on the agent's desk. To obtain the key and the secret, see: [Google Maps Platform](#)

Google Maps Key

Google Maps Secret *

Google Maps Language *

Google Maps Height *

 pixels

Google Maps Width *

 pixels

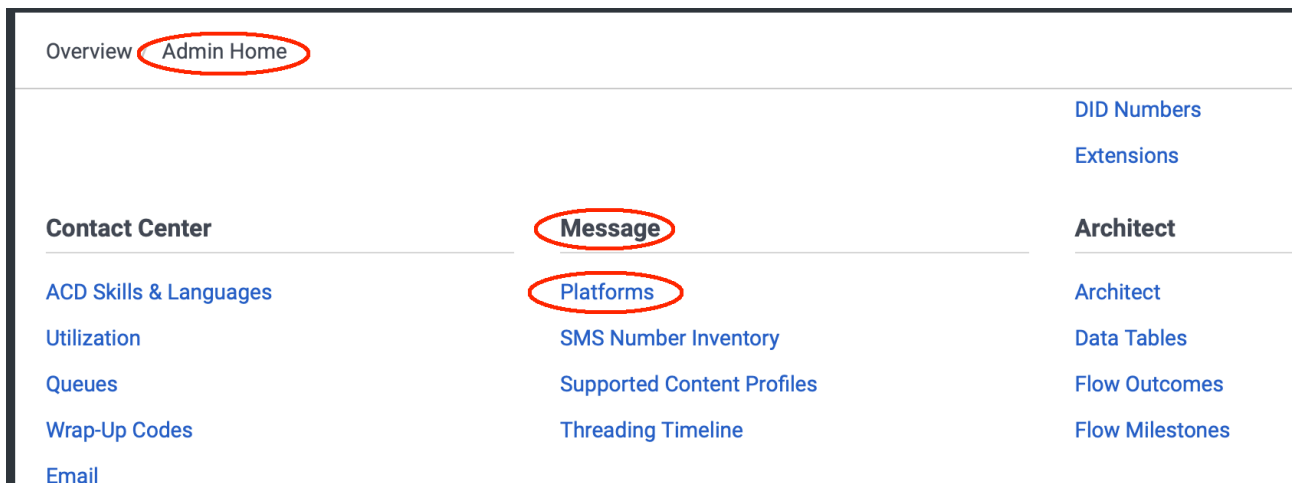
Google Maps Zoom Level *

* optional

Additional Configuration in Genesys Cloud CX

Once the Tenant is saved, a new Platform will appear in Genesys Cloud CX Administrator.

Connect to your Genesys Cloud CX Administration console, and go to the Message / Platforms section:



You should see a new Open Messaging Platform with the same name as the one you entered in Universal Messaging Config:

Message / Platforms

Platforms

- SMS Number Inventory
- Supported Content Profiles
- Threading Timeline

Open Messaging

Active

Name
OpenMessaging Integration Test

Supported Content
default

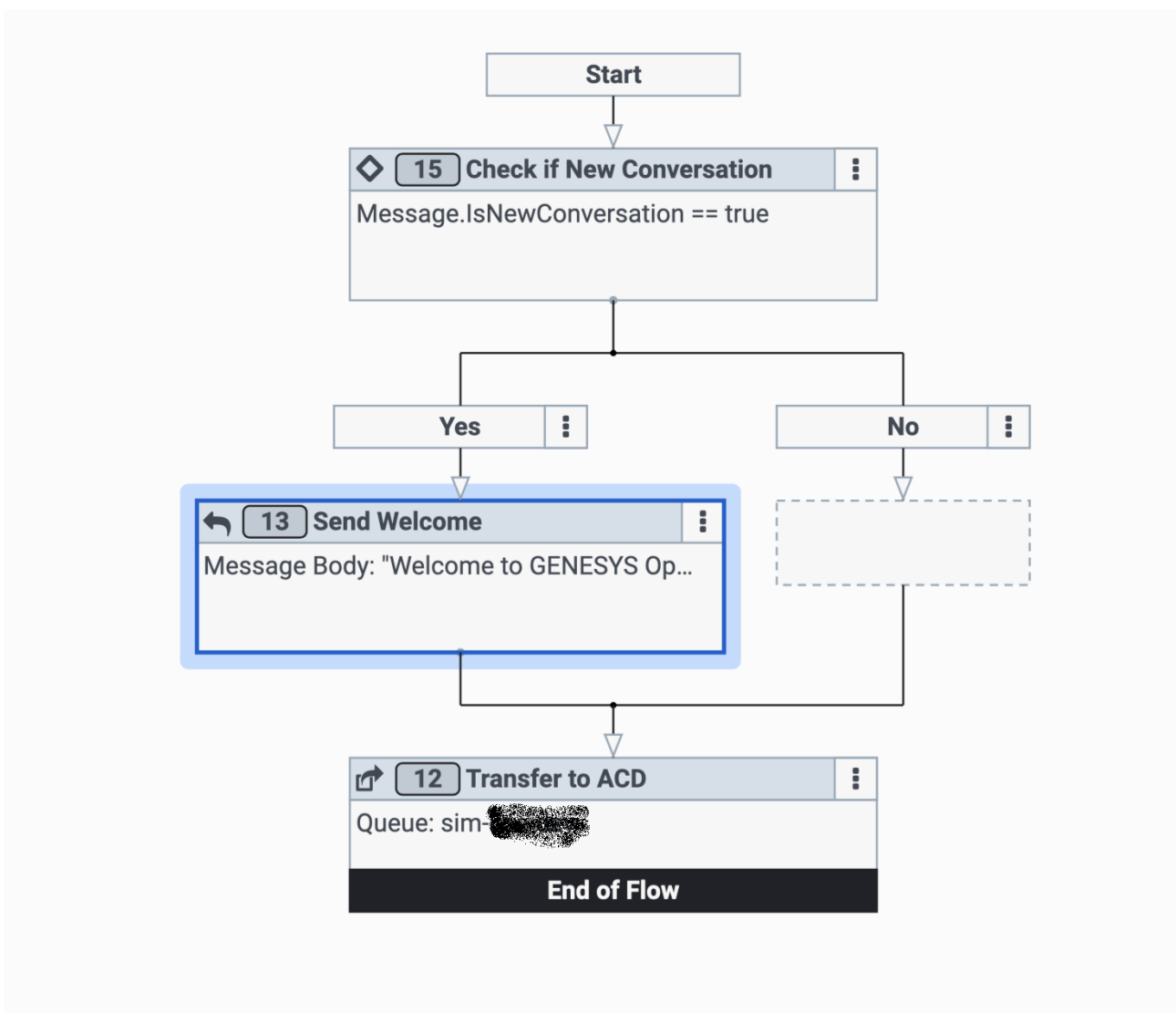
Outbound Notification Webhook URL
https://acme.ngrok.io/openmessaging

Outbound Notification Webhook Signature Secret Token
Hidden

Save **Cancel** **Delete**

There is nothing to do here, just verify the values entered in Universal Messaging Config were carried over properly.

Then create a new Inbound Message Flow in Architect. Here is an example (called "TestOpenMessaging"):



Once you publish it, it will appear in the section Routing / Message Routing of your Administration console:

Overview / **Admin Home**

Predictive Engagement

- Live Now
- Segments
- Outcomes
- Action Maps
- Action Library
- Global Settings
- Visitor Activity Report

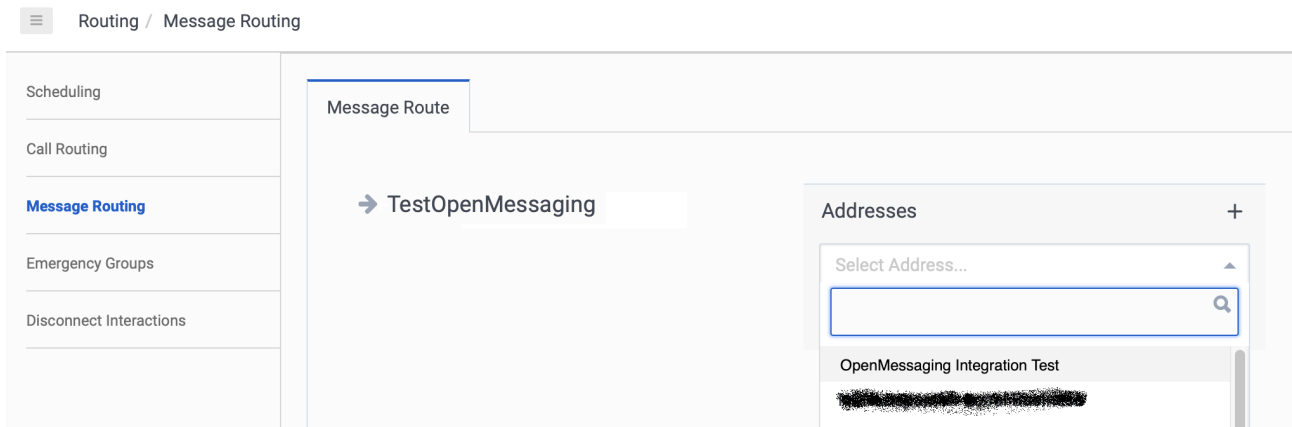
Routing

- Scheduling
- Call Routing
- Message Routing**
- Emergency Groups
- Disconnect Interactions

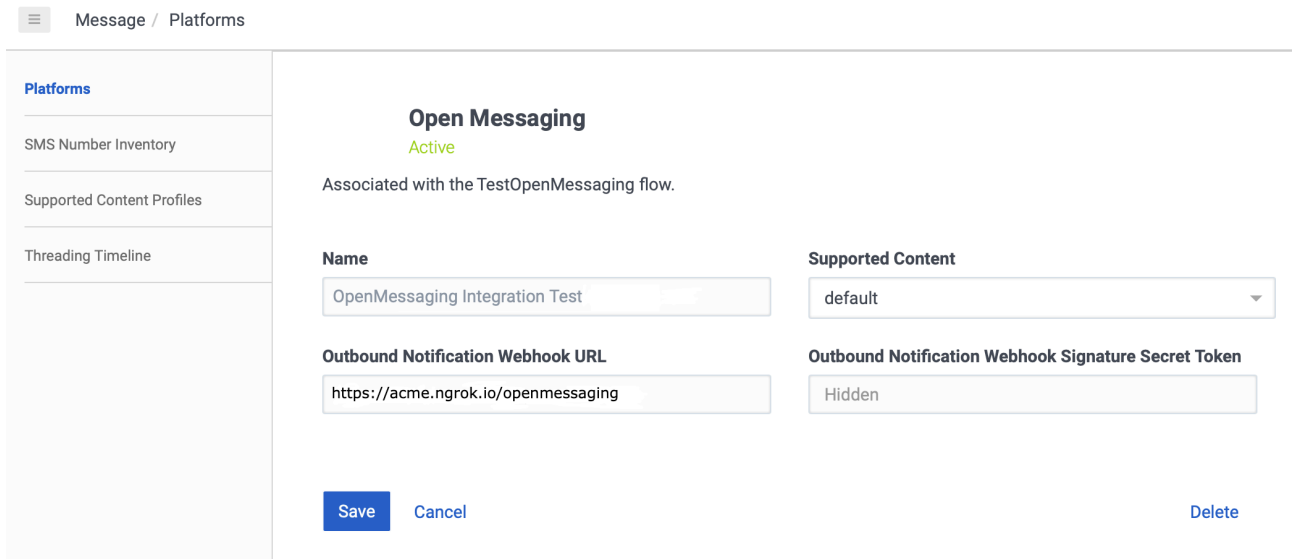
Troubleshooting

- Audit Viewer

Select the flow that was created with Architect, and assign the address from the Open Messaging Platform:



Once, this is done, if you go back to the Open Messaging Platform, you will see it is "linked" to the flow:



Genesys Cloud CX is now configured to send Open Messaging conversation to Universal Messaging.

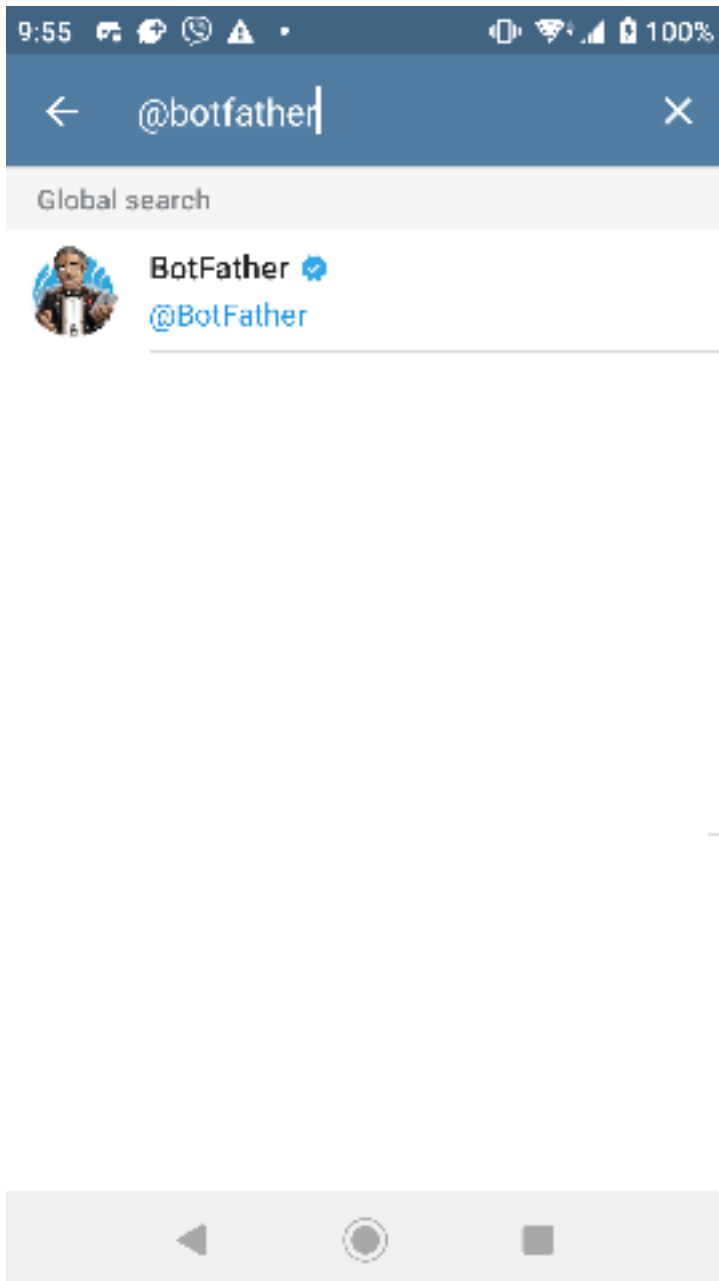
Note: If you want to link several Tenants to the same Inbound Message Flow, just choose distinct Open Messaging Names and add them to the same Message Route.

Messaging

Telegram

To connect to Telegram, you create a new Bot with Telegram's BotFather.

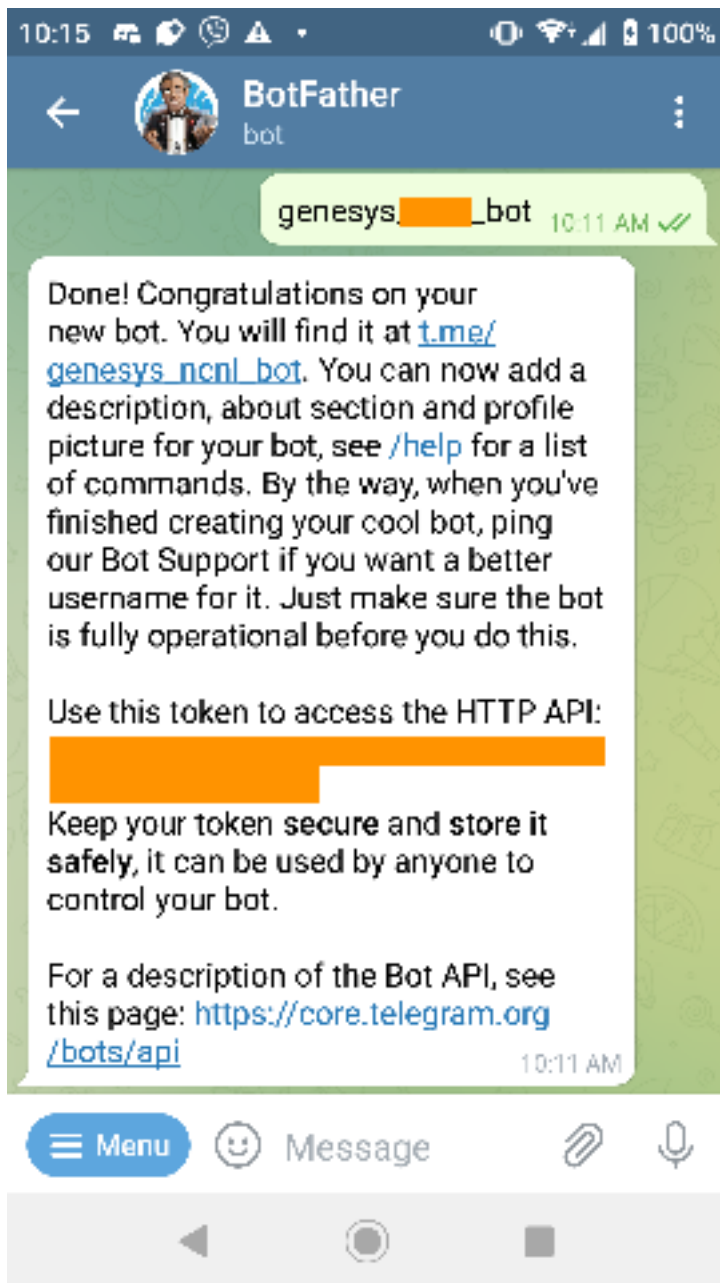
First, you need to *talk* to the BotFather, open the search and type `@BotFather` :



Once connected, start a new conversation (hit the `START` button) and create your new Bot by simply typing:

```
/newbot
```

This starts a dialog where you need to enter the name of your new Bot and its username. At the end of the dialog, the BotFather gives you an API Token:



Copy that API Token and enter it in the Universal Messaging Config:

Tenants 0

Tenant Name

Tenant

General API Authentication CX Platform Commander Messaging

Messaging Type

Telegram

API Token

51[REDACTED]2603:[REDACTED]uONIXwfES8[REDACTED].KrwL4

Webhook

https://[REDACTED]/telegram

Options:

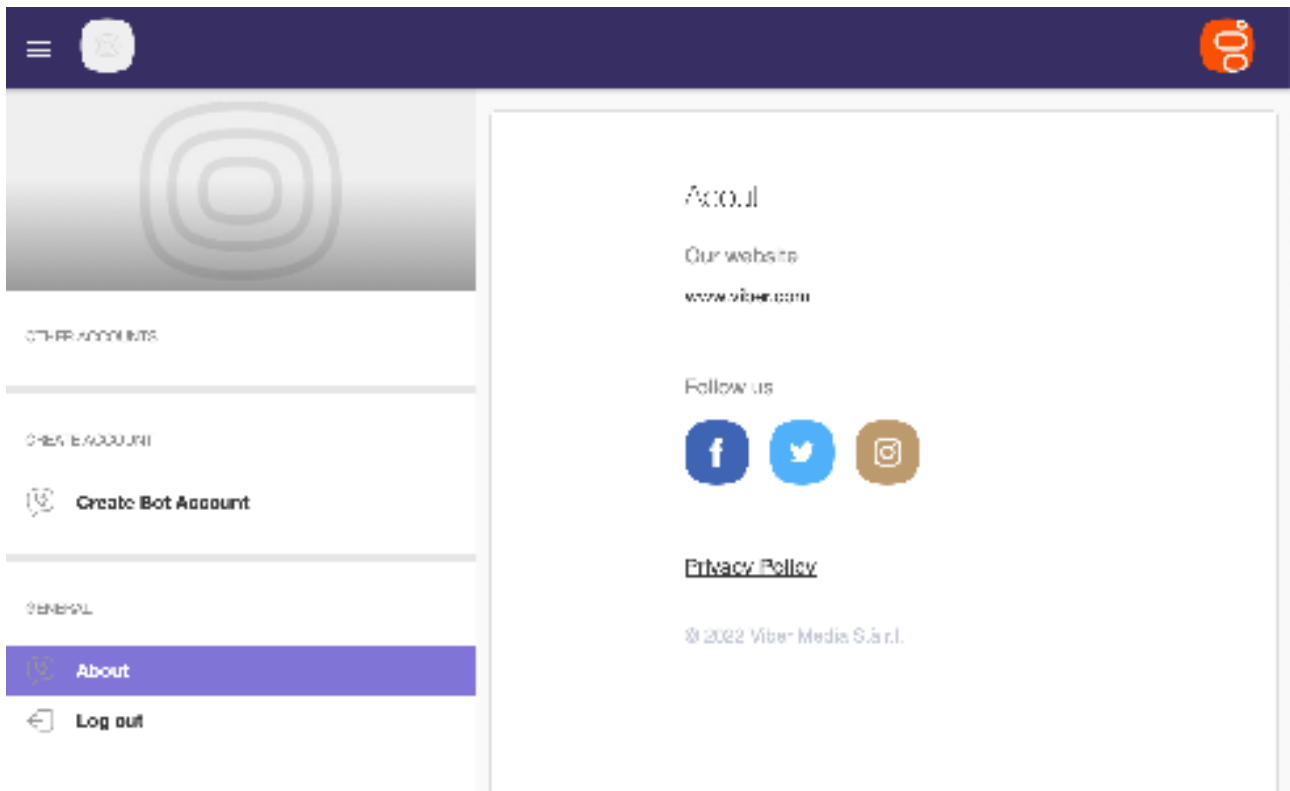
- Show System Messages
- Prefix Agent Name

Guest users will need to start a conversation with the Bot by mentioning its username prepended with a @ .

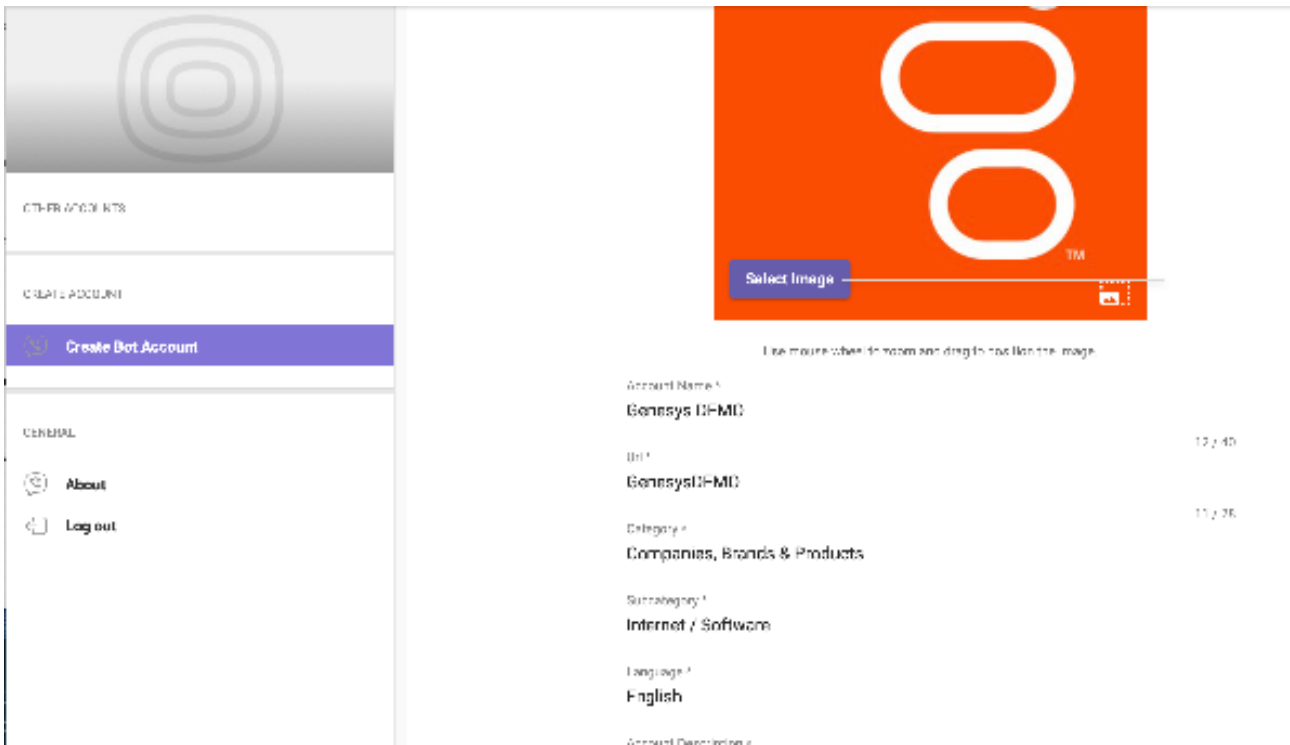
Viber

First create a [Viber Partner Account](#) and create a Viber Chatbot.

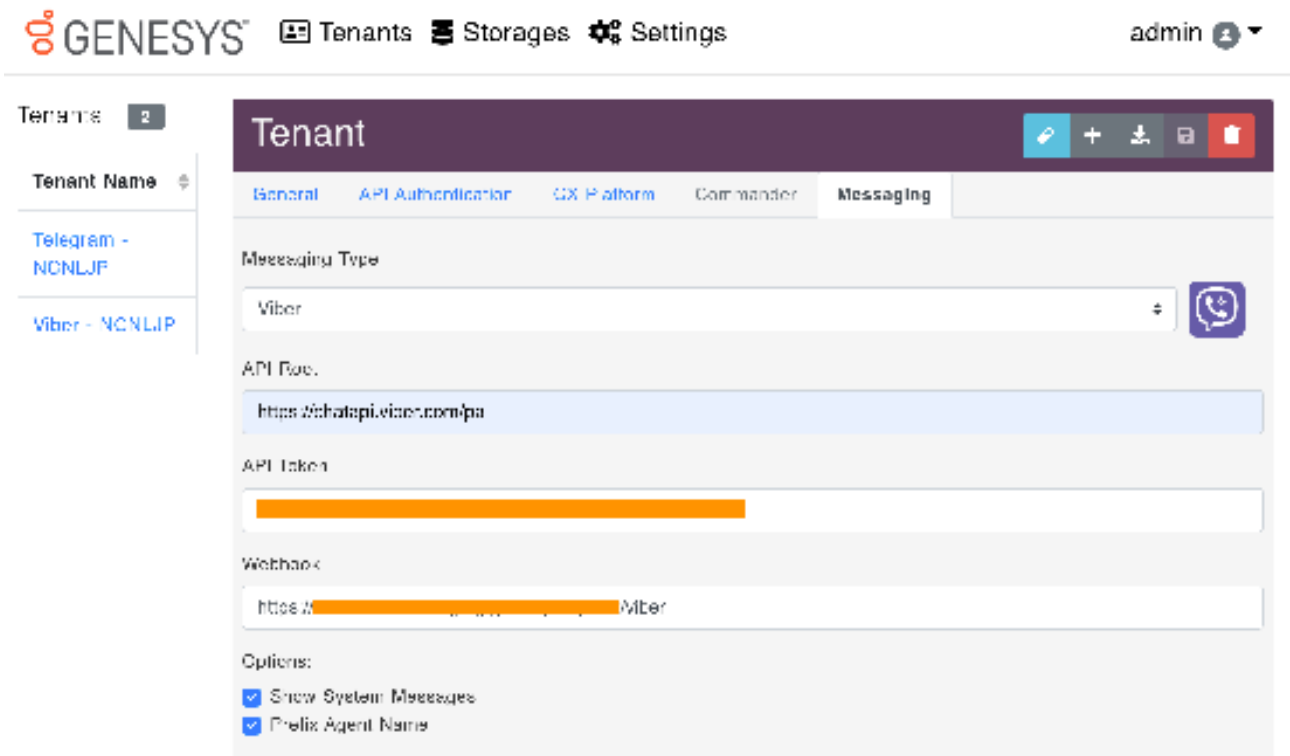
Then, log on your account at <http://partners.viber.com>:



And Create a Bot Account.




Upon creation, you will receive a Chat API Token that you will need to enter in the Universal Messaging Config:



Changing the administrator password

To change the administrator password (after initial installation), you just need to connect to the configuration website and use the "user" menu on the top-right:

admin  ▼

Change Password...

Log out...

Then simply enter the current password followed by the new password twice:

Please enter a new password...

Username

admin

Current Password

●●●●●●



New Password

●●●●●●



New Password (again)

●●●●●●



OK

Cancel

Note: The new password must be a strong password. We use the [zxcvbn](#) method to check the strength of the new password. That means, the password really has to be complex... no more simple "S3cr3t P@ssw0rd!"

Chapter 4

Using the REST API

The Universal Messaging application is usually configured by its config website. In addition, a REST API can also be used that allow batch scripts to create storages, tenants, etc.

That REST API is available at the FQDN of your deployment under the path `/api/v1`. For example:

```
https://www.acme.org/api/v1
```

The REST API [OpenAPI](#) specification is available at `/api/v1/openapi` and the API can be experienced via a [swagger-ui](#) available at: `/api/v1/openapi-ui`.

Note: To make sure the `/api/v1/openapi-ui` points to the Universal Messaging API, make sure the `api_root` property was set to the proper FQDN during the Helm deployment:

```
api:
  admin:
    password: s1ms3cr3t
  api_root: https://www.acme.org
```

To use the API, you must first authenticate:

```
curl https://www.acme.org/api/v1/auth/login \
-X POST \
-H "Content-Type: application/json" \
-d '{"user": "admin", "password": "s3cr3t}"'
```

The response will contain a token:

```
{
  "type": "Bearer",
  "token": "1234567876543wsdfghjgfvxqsq1234567ythgv"
}
```

Include the token in all future requests:

```
curl https://www.acme.org/api/v1/storages \
-H "Authorization: bearer 1234567876543wsdfghjgfvxqsq1234567ythgv"
```

REST API for IVRs

The Universal Messaging API can also be used by IVR engines, such as Genesys Cloud CX Architect and Genesys PureCloud Interaction Attendant.

When guests call your IVR it can be desirable to send them some messages to their Social Media accounts. With the Universal Messaging API, this is now possible.

Note that you will need to ask or store the guest's Social Media identifier.

A noticeable exception is LINE. If you subscribe to their Phone Number Push option (and filled in the configuration properly in the config website), you can use the guests (fully qualified) mobile numbers directly from the IVR. Make sure to contact LINE for to be able to use that option.

To send messages, the IVR will need to use the Tenant's Authentication defined in the config website:

```
curl https://www.acme.org/api/v1/messages \
-X POST \
-u tenant_apiuser:tenant_apipassword \
-H "Content-Type: application/json" \
-d \
'{
  "tenantId": "259bb4e2-f7e5-4b06-9a51-c528f412404b",
  "message": "Hello from the IVR!"
}'
```

Where `123456789abcd` is the Guest's identifier with the Social Media associated with the `tenantId` given in the JSON payload.

You can also use the Tenant's name in the payload:

```
curl https://www.acme.org/api/v1/messages \
-X POST \
-u tenant_apiuser:tenant_apipassword \
-H "Content-Type: application/json" \
-d \
'{
  "tenant": "Tenant with Social Media X and Genesys Cloud CX",
  "userId": "123456789abcd",
  "message": "Hello from the IVR!"
}'
```

It is also possible to send more than one message to the guest:

```
curl https://www.acme.org/api/v1/messages \
-X POST \
-u tenant_apiuser:tenant_apipassword \
-H "Content-Type: application/json" \
-d \
'{
  "tenantId": "259bb4e2-f7e5-4b06-9a51-c528f412404b",
  "userId": "123456789abcd",
  "messages": [
    "Hello from the IVR!",
    { "type": "sticker", "stickerId": "123345", "packageId": "456" }
  ]
}'
```

Of course, in the last example, the Social Media need to support Stickers.

If the Social Media is LINE and you have subscribed to the LINE Switcher API, it is possible to also send a request to "switch" the User to another LINE Bot Destination (please consult your LINE provider about this option):

```
curl https://www.acme.org/api/v1/users/+819012345678/switch \
-X POST \
-u tenant_apiuser:tenant_apipassword \
-H "Content-Type: application/json" \
-d \
'{
  "tenantId": "259bb4e2-f7e5-4b06-9a51-c528f412404b"
}'
```

Chapter 5

Kubernetes Fundamentals

The best and most efficient way of deploying Kubernetes is to use a Cloud vendor. Most of the time, they will manage the Kubernetes control plane for you (they might even offer that for free), upgrade the cluster automatically, make sure it works, and even have some node-level scalability (meaning adding Kubernetes nodes automatically as needs grow).

Here we will describe how to deploy Kubernetes in the most common of the Cloud vendors as well as more manual methods.

Disclaimer: These are given only to help, make sure you read the documentation of your chosen platform for fine-tuning your Kubernetes cluster.

Google Cloud Platform (GKE)

You can use the [Google Cloud Shell](#) or install the [Google Cloud SDK](#) for your platform, then initialize it:

```
gcloud init
gcloud auth login
```

- Create a project on the [Google Console](#),
- Add [Billing](#) to the project,
- Enable the StackDriver Logging API on the [Google Console](#)
- Enable Kubernetes on your [Google Cloud console](#).

And create your Kubernetes cluster:

```
gcloud container clusters create messaging-cluster \
  --cluster-version=latest \
  --zone asia-northeast1-a
```

By default, 3 worker nodes are created and Role Base Access Control (RBAC) is enabled. Each node is a `n1-standard-1` (1 vCPU, 3.75 GB Memory) (See [Pricing](#) and [Calculator](#))

If you prefer to let Google handle everything, you can create the cluster in "autopilot" mode:

```
gcloud container clusters create-auto messaging-cluster
```

You can check the status of the nodes with:

```
gcloud compute instances list
```

If you do not have `kubectl` installed already, go ahead and install it:

```
gcloud components install kubectl
```

Then download the Kubernetes configuration for `kubectl` :

```
gcloud container clusters get-credentials messaging-cluster
```

Microsoft Azure Container Service (AKS)

You can use the [Azure Cloud Shell](#) or install the Azure CLI on your machine:

- On bash:

```
brew install azure-cli
```

On Debian Linux:

```
sudo apt install azure-cli
```

On Red Hat and CentOS:

```
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
sudo tee /etc/yum.repos.d/azure-cli.repo &>/dev/null <<EOM
[azure-cli]
name=Azure CLI
baseurl=https://packages.microsoft.com/yumrepos/azure-cli
enabled=1
gpgcheck=1
gpgkey=https://packages.microsoft.com/keys/microsoft.asc"
EOM
sudo dnf install azure-cli
```

- On Windows, via [chocolatey](#):

```
chocolatey install azure-cli
```

- Or via PowerShell:

```
Invoke-WebRequest -Uri https://aka.ms/installazurecliwindows -OutFile .\AzureCLI.msi
Start-Process msiexec.exe -Wait -ArgumentList '/I AzureCLI.msi /quiet'
rm .\AzureCLI.msi
```

You can follow the [Kubernetes walkthrough](#) from Microsoft themselves. Here is a shortlist from their site.

First Login to your Azure account:

```
az login
```

If needed, enable Azure Container Services (AKS) and a few other things on your Azure Subscription (this can take some time!):

```
az provider register -n Microsoft.Compute
az provider register -n Microsoft.Storage
az provider register -n Microsoft.Network
az provider register -n Microsoft.ContainerService
az provider register -n Microsoft.OperationsManagement
az provider register -n Microsoft.OperationalInsights
```

If you are not sure AKS is delivered in your region, you can check what is available:

```
az aks get-versions --location 'japan east' \
  --query 'orchestrators[].orchestratorVersion'
```

If all is fine, create a resource group:

```
az group create --name genesys --location japaneast
```

Then, create your cluster:

```
az aks create --resource-group genesys \
  --name messaging-cluster \
  --node-count 3 \
  --enable-addons monitoring \
  --generate-ssh-keys
```

If you want specific Virtual Machine size, you should add a `---node-vm-size` .

If you want to load your own ssh key, use `--ssh-key-value /path/to/key.pub`

If you do not have `kubectl` installed already, go ahead and install it:

```
az aks install-cli
```

Then download the Kubernetes configuration for `kubectl` :

```
az aks get-credentials --resource-group genesys --name lis-cluster
```

Then verify you can get to the Kubernetes Cluster:

```
kubectl get nodes
```

Amazon Elastic Container Services for Kubernetes (EKS)

Install the AWS CLI on your machine then install eksctl by following its [user guide](#):

- On MacOS:

```
brew install awscli
brew tap weaveworks/tap
brew install weaveworks/tap/eksctl
```

- On Windows:

```
chocolatey install awscli
```

- On Linux:

```
curl -sSLO "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip"
unzip awscli-exe-linux-x86_64.zip
sudo ./aws/install
curl -sSL \
  "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_${uname -s}_amd64.tar.gz" \
  | tar xz
sudo mv ./eksctl /usr/local/bin
```

Configure AWS with your Access Key ID and Secret Access Key (ask your AWS Administrator for these values):

```
aws configure
```

If you use an Multi-Factor Authentication device, before executing any aws/eksctl command, you should obtain temporary credentials:

```
function aws-auth() {
  token_code=$1
  unset AWS_ACCESS_KEY_ID AWS_SECRET_ACCESS_KEY AWS_SESSION_TOKEN
  aws_identity=$(aws sts get-caller-identity --output json)
  aws_account=$(echo "$aws_identity" | jq -r .Account)
  aws_user=$(echo "$aws_identity" | jq -r .Arn | cut -d/ -f2)
  aws_arn="arn:aws:iam::${aws_account}:mfa/${aws_user}"
  creds=$(aws sts get-session-token --serial-number "$aws_arn" --token-code $token_code --output json)
  export AWS_ACCESS_KEY_ID=$(echo "$creds" | jq -r .Credentials.AccessKeyId)
  export AWS_SECRET_ACCESS_KEY=$(echo "$creds" | jq -r .Credentials.SecretAccessKey)
  export AWS_SESSION_TOKEN=$(echo "$creds" | jq -r .Credentials.SessionToken)
  export AWS_TOKEN_EXPIRATION=$(echo "$creds" | jq -r .Credentials.Expiration)
  unset aws_identity creds aws_account aws_user aws_arn
}
aws-auth 123456
```

The token code comes from your MFA Device. make sure this function belongs to your bashrc/zshrc, so it can modify the current environment (if it is a script, that will not work)

You can download this function [there](#) and its Powershell equivalent [there](#).

Fargate

The simplest way to deploy a Kubernetes cluster is through Fargate as it manages worker nodes automatically, including scaling them up and down.

Create the cluster:

```
eksctl create cluster \
  --name messaging-cluster \
  --region ap-northeast-1 \
```

```
--fargate \
--alb-ingress-access
```

After 15 to 25 minutes (for both Fargate and Managed Nodes), on average, your Kubernetes cluster is ready to use.

Finally, you can query the newly created cluster:

```
kubectl get nodes
```

We will also need to tell Fargate it should also host the Kubernetes namespace for our application:

```
eksctl create fargateprofile \
  --cluster messaging-cluster \
  --name fp-messaging \
  --namespace messaging
```

You can enable AWS Cloudwatch logging for the control plane, if you wish, by running this:

```
eksctl utils update-cluster-logging \
  --cluster messaging-cluster \
  --region ap-northeast-1 \
  --enable-types all \
  --approve
```

Note: This implies some cost for the log storage. See <https://eksctl.io/usage/cloudwatch-cluster-logging>.

Next, we need to install the [Amazon EFS CSI Driver](#) so we can create `PersistentVolume` objects with Fargate.

If the cluster does not have an OIDC provider, yet:

```
eksctl utils associate-iam-oidc-provider \
  --cluster messaging-cluster \
  --region ap-northeast-1 \
  --approve
```

AWS EFS CSI Driver for Persistent Volumes

Create a new IAM policy to allow the CSI driver to use the AWS API on your behalf:

```
oidc=$(aws eks describe-cluster \
  --name messaging-cluster \
  --query cluster.identity.oidc.issuer \
  --output text \
)
curl -sSLO \
  https://raw.githubusercontent.com/kubernetes-sigs/aws-efs-csi-driver/\
v1.3.2/docs/iam-policy-example.json
aws iam create-policy \
  --policy-name AmazonEKS_EFS_CSI_Driver_Policy \
  --policy-document file://iam-policy-example.json
```

Create an IAM role attached to that new policy:

```
policy=$(aws iam list-policies --output json | \
  jq -r '.Policies[]|select(.PolicyName=="AmazonEKS_EFS_CSI_Driver_Policy")|.Arn' \
)
eksctl create iamserviceaccount \
  --name efs-csi-controller-sa \
  --cluster messaging-cluster \
  --namespace kube-system \
  --region ap-northeast-1 \
  --attach-policy-arn $policy \
  --override-existing-serviceaccounts \
  --approve
```

Deploy the CSI Driver:

```
kubectl create \
  -f https://raw.githubusercontent.com/kubernetes-sigs/aws-efs-csi-driver/\
  master/deploy/kubernetes/base/csidriver.yaml
```

Create a security group and an inbound rule for NFS:

```
vpcid=$(aws eks describe-cluster \
  --name messaging-cluster \
  --query cluster.resourcesVpcConfig.vpcId \
  --output text \
)
range=$(aws ec2 describe-vpcs \
  --vpc-ids $vpcid \
  --query "Vpcs[].CidrBlock" \
  --output text \
)
sec_group=$(aws ec2 create-security-group \
  --group-name messaging-efs-security-group \
  --description "EFS Security Group for Universal Messaging" \
  --vpc-id $vpcid \
  --output text \
)
aws ec2 authorize-security-group-ingress \
  --group-id $sec_group \
  --cidr $range \
  --protocol tcp \
  --port 2049
```

Create an Amazon EFS file system:

```
fs_id=$(aws efs create-file-system \
  --region ap-northeast-1 \
  --performance-mode generalPurpose \
  --query FileSystemId \
  --output text \
)
```

And a mount target for Amazon EFS for each subnet member of the Fargate Profile:

```
subnets=( $(eksctl get fargateprofiles --cluster messaging-cluster -- json | \
  jq -r '.[ ] | select(.name=="fp-messaging") | .subnets[]' \
) )
for subnet in ${subnets[@]}; do
  aws efs create-mount-target \
    --file-system-id $fs_id \
    --subnet-id $subnet \
    --security-groups $sec_group
done
```

If there is another default storage class, we first need to depromote it:

```
kubectl patch storageclass gp2 \
  --patch '{"metadata":{"annotations":{"storageclass.kubernetes.io/is-default-class":"false"}}}'
```

Now, we can create the Kubernetes StorageClass and make it the default storage class:

```
kubectl create -f - <<EOM
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: efs-sc
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: efs.csi.aws.com
parameters:
```



```
provisioningMode: efs-ap
fileSystemId: ${fs_id}
directoryPerms: "700"
```

EOM

When creating the config.yaml, we will need to ask the charts to create the PersistentVolume resources, since Fargate does not support [Dynamic Volume Provisioning](#), yet. Here we show the necessary configuration for a few micro-services:

```
global:
  storageClass: efs-sc
  storageDriver: efs.csi.aws.com
  storageHandle: ${fs_id} # Replace this with the actual fs_id from your cluster
api:
  ...
  redis:
    volumePermissions:
      enabled: true
    master:
      persistence:
        ...
        selector:
          matchLabels:
            owner: api
          volume: { create: true }

gcloudcx-connector:
  ...
  redis:
    volumePermissions:
      enabled: true
    master:
      persistence:
        ...
        selector:
          matchLabels:
            owner: gcloudcx-connector
          volume: { create: true }

rabbitmq:
  ...
  persistence:
    ...
    selector:
      matchLabels:
        owner: rabbitmq
      volume: { create: true }
  # See: https://docs.bitnami.com/general/how-to/troubleshoot-helm-chart-issues/#permission-errors-when-enablin
  volumePermissions:
    enabled: true
  # workaround for https://github.com/bitnami/bitnami-docker-rabbitmq/issues/86
  ulimitNofiles: ""
```

Deploy the ALB Ingress Controller

The next step is to deploy the AWS Load Balancer (ALB) to your new Kubernetes cluster.

Create a new IAM policy to allow ALB to use the AWS API on your behalf:

```
curl -sSL0 \
  https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balancer-controller/\
  v2.2.0/docs/install/iam_policy.json
aws iam create-policy \
```

```

    --policy-name AWSLoadBalancerControllerIAMPolicy \
    --policy-document file://iam_policy.json
rm iam_policy.json

```

If the cluster does not have an OIDC provider, yet:

```

eksctl utils associate-iam-oidc-provider \
  --cluster messaging-cluster \
  --region ap-northeast-1 \
  --approve

```

Create an IAM role attached to that new policy:

```

policy=$(aws iam list-policies --output json | \
  jq -r '.Policies[]|select(.PolicyName=="AWSLoadBalancerControllerIAMPolicy")|.Arn' \
)
eksctl create iamserviceaccount \
  --name      aws-load-balancer-controller \
  --cluster   messaging-cluster \
  --namespace kube-system \
  --attach-policy-arn $policy \
  --override-existing-serviceaccounts \
  --approve

```

Install ALB via Helm:

```

helm repo add eks https://aws.github.io/eks-charts
helm repo update
kubectl apply -k "github.com/aws/eks-charts/stable/aws-load-balancer-controller/crds?ref=master"
vpcid=$(aws eks describe-cluster \
  --name messaging-cluster \
  --query cluster.resourcesVpcConfig.vpcId \
  --output text \
)
helm upgrade -i aws-load-balancer-controller eks/aws-load-balancer-controller \
  --namespace kube-system \
  --set clusterName=messaging-cluster \
  --set serviceAccount.create=false \
  --set serviceAccount.name=aws-load-balancer-controller \
  --set region=ap-northeast-1 \
  --set vpcId=$vpcid
kubectl rollout status deployment \
  --namespace kube-system \
  aws-load-balancer-aws-load-balancer-controller
kubectl get deployments.apps --namespace kube-system

```

The last two commands just check if ALB is deployed correctly.

When deploying the Universal Messaging application, you should configure the ingress as follows in the config.yaml:

```

ingress:
  enabled: true
  className: alb
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/scheme: internet-facing

```

Also, all Kubernetes service types must be `NodePort` for ALB to route HTTP requests. In the config.yaml, configure the services as follows (here we show only the Universal Messaging API and config configurations):

```

api:
  service:
    type: NodePort
    annotations:

```

```

alb.ingress.kubernetes.io/healthcheck-port: "32000"
alb.ingress.kubernetes.io/healthcheck-path: "/healthz/readiness"

config:
  service:
    type: NodePort
    annotations:
      alb.ingress.kubernetes.io/healthcheck-port: "3000"
      alb.ingress.kubernetes.io/healthcheck-path: "/"
      alb.ingress.kubernetes.io/success-codes: "200,302"

```

You can find more details about ALB Ingress on [AWS site](#).

Delete the cluster

If you need to delete the cluster, run this:

```

eksctl delete cluster \
  --name messaging-cluster \
  --region ap-northeast-1

```

Docker for Desktop

This is the simplest when you want to develop and/or learn and you run on Windows or macOS:

1. Just install Docker for Windows or Docker for Mac.
2. Turn on Kubernetes in the Docker settings.

You are done!

Check it:

```
kubectl get nodes
```

Note: Docker for Desktop's Kubernetes is fairly "naked" by default. This means there is no Ingress Controller, no Registry, no Logging stack. You have to install these by yourself, if you need them.

For example: If you want a local registry within the Kubernetes cluster. First install [Helm](#) and then deploy the registry chart:

```

helm install stable/docker-registry \
  --namespace kube-system \
  --set persistence.enabled=true \
  --set service.type=NodePort \
  --set service.nodePort=32000

```

To install an Ingress Controller, the best is to follow its [installation guide](#):

```

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/\
controller-v1.0.0/deploy/static/provider/cloud/deploy.yaml

```

When deploying an ingress you will need to add the ingress class "nginx" in the annotation:

```

metadata:
  annotations:
    kubernetes.io/ingress.class: nginx

```

With Universal Messaging's Helm charts, you simply need to add the class in your config.yaml:

```

ingress:
  className: "nginx"

```

About the Logging stack, I have become a big fan of [fluentd](#), [Digital Ocean](#) made an excellent documentation about deploying it.

MicroK8s

MicroK8s is a development/test Kubernetes deployment on Windows, MacOS, and Ubuntu. It has the advantage of simplicity and easiness. It is perfect for Development and Learning (not for production!)

To install MicroK8s, simply execute:

```
sudo snap install microk8s --classic
```

After a short while, you Kubernetes cluster is ready.

You will typically want to enable some services, like:

```
sudo microk8s.enable rbac dns registry ingress storage helm3
snap alias microk8s.kubectl kubectl
snap alias microk8s.helm3 helm
sudo iptables -P FORWARD ACCEPT
sudo DEBIAN_FRONTEND=noninteractive apt-get install -y iptables-persistent
```

Also check that `:::1` does not resolve into `localhost` in `/etc/hosts`.

You might have to configure the DNS forwarder in `coredns` as it generally points to 8.8.8.8 and 8.8.4.4 and that might not be suitable in your environment, see <https://microk8s.io/docs/addon-dns>.

If `microk8s` is on the same host, you are done.

If you want to access `microk8s` remotely from your desktop, you need to install `kubectl` and `helm` locally and configure remote access.

To add the cluster configuration to your local kubectl, you can grab its config, from your machine:

```
ssh myuser@kubernetes-host microk8s.config view > ~/.kube/myk8s
```

Do not forget to properly merge `myk8s` and `~/.kube/config` if you don't want to set the environment variable `KUBECONFIG`.

To be able to push docker images to your cluster, you should also add its IP or FQDN to the insecure registries of your own docker configuration:

On Linux, you would edit `/etc/docker/daemon.json` and add:

```
{
  "insecure-registries": [ "kubernetes-host:32000" ]
}
```

Where `kubernetes-host` should be replaced by either the hostname or its IP address.

On Docker for Windows and Docker for Mac, you would edit add the registry in the Preferences dialog.

For more informations about Microk8s configuration and usage, see: <https://microk8s.io/docs>

Manual Deployment on Virtual Machines/Bare Metal

Sometimes, the usual Cloud vendors do not have any support for Containers in your region. In that case, the solution is to build the Kubernetes cluster manually in virtual machines or bare metal (We will use the term Virtual Machine in this paragraph, but, unless mentioned, everything applies to bare metal machines as well). While you get full control, this is really a lot more complex and it is very easy to make mistakes...

As an example, here we will build a cluster with 1 master and 3 worker nodes. You should change these numbers to your own needs.

First, get 4 virtual machines from your Cloud vendor or your own data center. I prefer going with Ubuntu 16.04 as it boasts a 4.x kernel already.

If you could not manually create partitions and have a swap, disable the swap on every node, afterwards:

```
sudo swapoff -a
sudo sed -i ' / swap /s/^/#/' /etc/fstab
```

Notes:

- Make sure the master node always gets the same IP address!!!!
- The following script will try to remove the swap if you forgot to do it.

On the Kubernetes master node, install all the software and start the cluster:

```
curl -sSL https://tinyclouds.org/ubuntu-prep-k8s | bash -s
```

At the end of the script, you will get a token, the master's IP address, and a certification hash. Copy these values and use them on each worker node:

```
curl -sSL https://tinyclouds.org/ubuntu-prep-k8s | bash -s -- \
  --join 123.456.78.9:6443 \
  --token 1046fd.a914436354c9c418 \
  --discovery-token-ca-cert-hash \
  sha256:af8b270b8745809dec666b8048663a087c3b9d9d6d8cbd6db62748503403f2ba
```

If you want to manage the cluster from your own machine, install `kubectl` and add the configuration from the master node:

```
scp myuser@123.456.78.9:.kube/config ~/.kube/config-mycluster
export KUBECONFIG="$HOME/.kube/config-mycluster:$HOME/.kube/config${KUBECONFIG+:$KUBECONFIG}"
kubectl config use-context mycluster
```

The name of the context can vary, check the actual values with `kubectl config get-contexts`. You should also add the export command to your `.bashrc` or `.zshenv`

Check the cluster:

```
kubectl get nodes
```

If you installed Kubernetes on Bare Metal or "Bare" Virtual Machines, you most probably will not have a Network Load-Balancer. This means all your Kubernetes `LoadBalancer` services will have their external IP set to `<pending>` without ever gaining an IP address.

The best is to install a Network Load-Balancer on your Kubernetes platform, such as [MetalLB](#):

```
kubectl apply \
-f https://raw.githubusercontent.com/google/metallb/v0.5.0/manifests/metallb.yaml
```

```
kubectl apply -f - << EOM
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
      - name: default
        protocol: layer2
        addresses:
          - x.y.z.1-x.y.z.100
EOM
```

Replace the address range with something that makes sense for your network.

If you do not have access to any Network Storage in your Cloud or if you run in your DataCenter, you will need to deploy a Kubernetes Storage. As an example, here we will deploy [Rook](#) over [Ceph](#):

```
ROOK_URL=https://raw.githubusercontent.com/rook/rook/release-0.7/cluster/examples/kubernetes
kubectl apply -f $ROOK_URL/rook-operator.yaml
kubectl apply -f $ROOK_URL/rook-cluster.yaml
kubectl apply -f $ROOK_URL/rook-storageclass.yaml
kubectl apply -f $ROOK_URL/rook-object.yaml
kubectl apply -f $ROOK_URL/rook-filesystem.yaml
kubectl patch storageclass rook-block -p \
  '{"metadata":{"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
```

Check for the proper version of rook there: <https://github.com/rook/rook/releases>

Chapter 6

Getting Helm

Maintaining a plethora of YAML scripts to deploy and maintain your application is progress, but ultimately it becomes a mess. Even if you maintain them through version control (like git).

Enter the Package Manager and Release Deployment tool of choice: HELM (<https://helm.sh>)

First you need to acquire [Helm](#).

On Windows, with [chocolatey](#):

```
choco install -y kubernetes-helm
```

On MacOS, using [Homebrew](#):

```
brew install kubernetes-helm
```

On Linux, using [snap](#):

```
sudo snap install helm
```

In case you do not have access to a package manager, you can still get Helm there: <https://github.com/kubernetes/helm/releases>

Chapter 7

Troubleshooting

Helm upgrade is stuck

Sometimes [Helm](#) fails in the middle of deploying a release. Usually this happens when the charts are misconfigured but can still be deployed.

After executing:

```
helm upgrade
```

It fails and you cannot execute it again, you keep getting:

```
$ helm upgrade ...  
ERROR: UPGRADE FAILED: another operation (install/upgrade/rollback) is in progress
```

Although, there is nothing in the release list:

```
$ helm list --namespace messaging  
NAME      NAMESPACE      REVISION      UPDATED STATUS  CHART      APP VERSION
```

The solution is to delete the information about that release in the Kubernetes cluster.

To do this, we need to understand where that information is stored. In Helm 3.0+, it is stored in some secrets of the namespace:

```
$ kubectl get secrets --namespace messaging --selector owner=helm  
NAME                                TYPE      DATA  AGE  
sh.helm.release.v1.demo.v1         helm.sh/release.v1  1      6d2h  
sh.helm.release.v1.demo.v2         helm.sh/release.v1  1      2d1h  
sh.helm.release.v1.demo.v3         helm.sh/release.v1  1      2d1h  
sh.helm.release.v1.demo.v4         helm.sh/release.v1  1      2d1h
```

The problematic release can be found like this:

```
$ kubectl get secrets --namespace messaging --selector owner=helm,status=pending-upgrade  
NAME                                TYPE      DATA  AGE  
sh.helm.release.v1.demo.v4         helm.sh/release.v1  1      2d1h
```

Delete that secret, and you can run helm upgrade again:

```
kubectl delete secrets sh.helm.release.v1.demo.v4  
helm upgrade ...
```