**Kong API Gateway and Konga dashboard installation on docker**

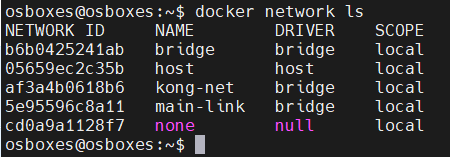
How to connect a Kong container to a Cassandra or PostgreSQL container.

Prerequisites:- Docker

Step1:- Create a Docker network

You will need to create a custom network to allow the containers to discover and communicate with each other. In this example kong-net is the network name, you can use any name.

$ docker network create kong-net



Step2: Start your database

If you wish to use a Cassandra container:

$ docker run -d --name kong-database \

--network**=**kong-net \

-p 9042:9042 \

cassandra:3

If you wish to use a PostgreSQL container:

$ docker run -d --name kong-database \

--network**=**kong-net \

-p 5432:5432 \

-e "POSTGRES\_USER=kong" \

-e "POSTGRES\_DB=kong" \

-e "POSTGRES\_PASSWORD=kong" \

postgres:9.6

Step3:- Prepare your database

Run the migrations with an ephemeral Kong container:

$ docker run --rm \

--network**=**kong-net \

-e "KONG\_DATABASE=postgres" \

-e "KONG\_PG\_HOST=kong-database" \

-e "KONG\_PG\_USER=kong" \

-e "KONG\_PG\_PASSWORD=kong" \

-e "KONG\_CASSANDRA\_CONTACT\_POINTS=kong-database" \

kong:latest kong migrations bootstrap

In the above example, both Cassandra and PostgreSQL are configured, but you should update the KONG\_DATABASE environment variable with either cassandra or postgres.

Note for Kong < 0.15: with Kong versions below 0.15 (up to 0.14), use the up sub-command instead of bootstrap. Also note that with Kong < 0.15, migrations should never be run concurrently; only one Kong node should be performing migrations at a time. This limitation is lifted for Kong 0.15, 1.0, and above.

Step 4:- Start Kong

When the migrations have run and your database is ready, start a Kong container that will connect to your database container, just like the ephemeral migrations container:

$ docker run -d --name kong \

--network**=**kong-net \

-e "KONG\_DATABASE=postgres" \

-e "KONG\_PG\_HOST=kong-database" \

-e "KONG\_PG\_USER=kong" \

-e "KONG\_PG\_PASSWORD=kong" \

-e "KONG\_CASSANDRA\_CONTACT\_POINTS=kong-database" \

-e "KONG\_PROXY\_ACCESS\_LOG=/dev/stdout" \

-e "KONG\_ADMIN\_ACCESS\_LOG=/dev/stdout" \

-e "KONG\_PROXY\_ERROR\_LOG=/dev/stderr" \

-e "KONG\_ADMIN\_ERROR\_LOG=/dev/stderr" \

-e "KONG\_ADMIN\_LISTEN=0.0.0.0:8001, 0.0.0.0:8444 ssl" \

-p 8000:8000 \

-p 8443:8443 \

-p 127.0.0.1:8001:8001 \

-p 127.0.0.1:8444:8444 \

kong:latest

Note:- We can user server’s IP instead of local host(127.0.0.1)

Step5 :- Use Kong

Kong is running:

$ curl -i http://localhost:8001/

How to setup Konga using Docker

Konga is an opensource powerful GUI that uses Kong Admin API, that makes managing of Kong quite literally…. a breeze.

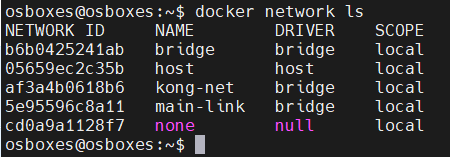
Today we are walking through setting up Konga using docker. We will be making use of PostgreSQL as our persistent database for Konga.

Step1:- First, we want to save PostgreSQL to a permanent folder, and to do that we will be creating a folder for this sole purpose

mkdir -p data

Step2:- Then the next step is to create a Docker network that will house both the PostgreSQL and Konga containers so that they can easily reach each other.  
P.S We could have them on the default network by default, but it is better to create network separation in case you have multiple docker images of the same type running. You do not want them clashing.

docker network create -d bridge main-link



Step3:- Then once done, we run the docker command to startup PostgreSQL using the alpine variant image and binding both the network and volume to the ones specified above. In addition to that, we set the default credentials for the database and set the name of the docker container

docker run --rm \

-e POSTGRES\_USER=konga \

-e POSTGRES\_DB=konga \

-e POSTGRES\_PASSWORD=konga \

-v data:/var/lib/postgresql/data \

--name postgres \

-d --network main-link \

postgres:9.6-alpine

Step4:- Then we need to seed the database with the Konga Schemas from the Konga image, we run the below before we start the Konga image proper.

docker run --network main-link --rm pantsel/konga:latest -c prepare -a postgres -u postgresql://konga:konga@postgres:5432/konga

Step5:- Once the Database has been seeded successfully, then you can start the docker image below.

docker run -p 1337:1337 \

--network main-link \

-e "TOKEN\_SECRET=ffffssf" \

-e "DB\_ADAPTER=postgres" \

-e "DB\_HOST=postgres" \

-e "DB\_PORT=5432" \

-e "DB\_USER=konga" \

-e "DB\_PASSWORD=konga" \

-e "DB\_DATABASE=konga" \

-e "NODE\_ENV=production" \

-d --name konga \

pantsel/konga

OR

docker run -p 1337:1337 --network=main-link -e "TOKEN\_SECRET=kong" -e "DB\_ADAPTER=postgres" -e "DB\_HOST=your-kong-database" -e "DB\_PORT=5432" -e "DB\_USER=kong" -e "DB\_PASSWORD=kong" -e "DB\_DATABASE=kong-database" -e "NODE\_ENV=production" -e "module=sails-postgresql" -e "ssl=true" --name konga pantsel/konga

docker run -p 1337:1337 \

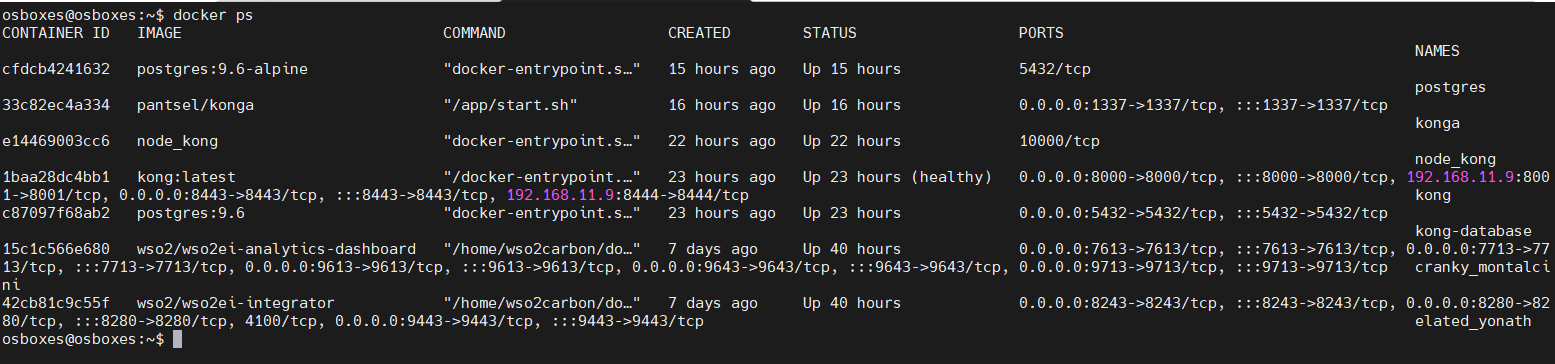
--link kong:kong \

--name konga \

--network main-link \

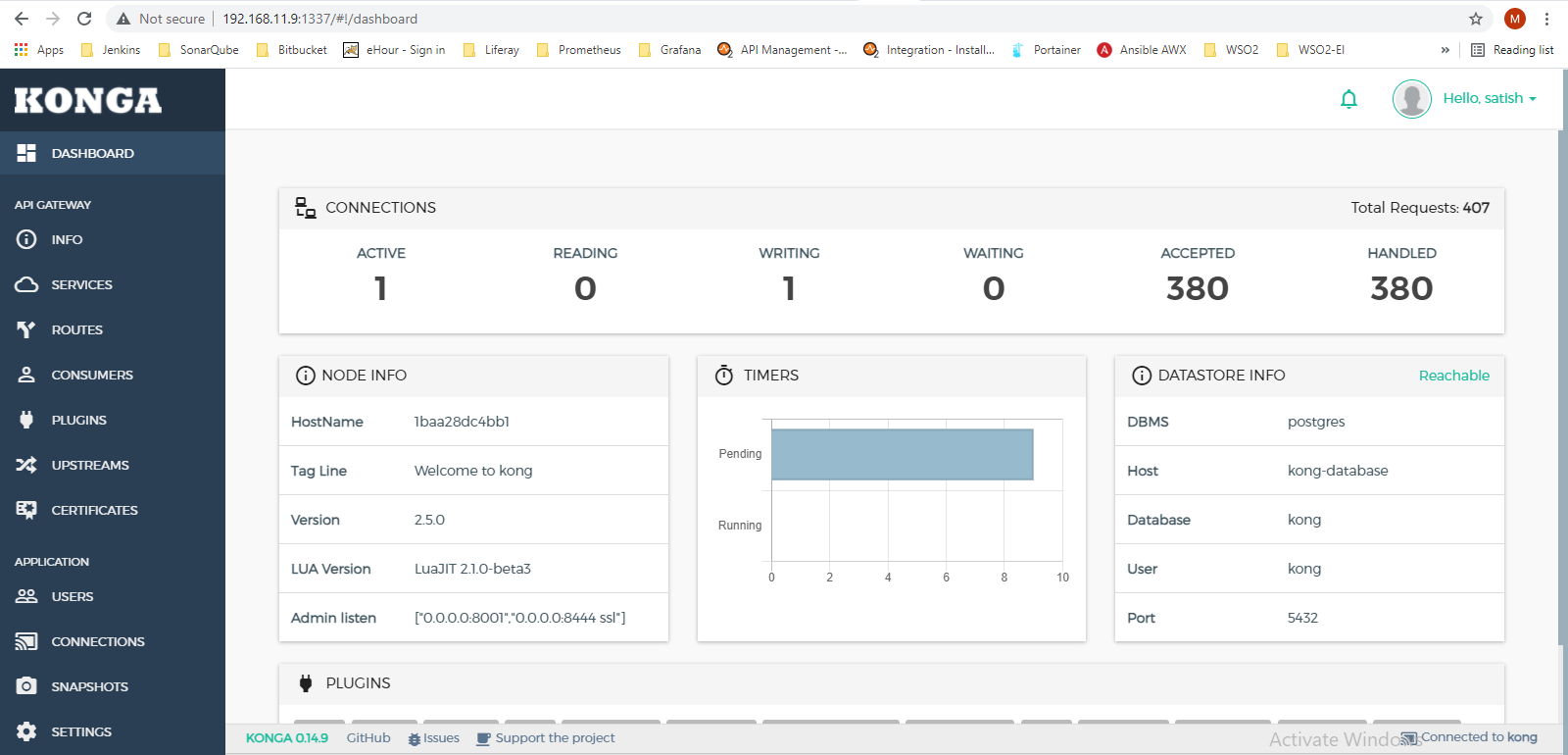
-e "NODE\_ENV=production" \

pantsel/konga



Step6:- Start using Konga dashboard in your browser with below IP

<http://192.168.11.9:1337/>



Reference URLs:-

1. For Kong API Manager installation

Link:- <https://docs.konghq.com/install/docker/>

1. For Konga dashboard installation

Link:- <https://www.dennisotugo.com/how-to-setup-konga-using-docker/>

OR

<https://frameworks.readthedocs.io/en/latest/framework/api/kongaDocker.html>

1. For Setting up Kong API Gateway

Link:- <https://mrkaran.dev/posts/setting-up-kong-part-1/>

<https://mrkaran.dev/posts/setting-up-kong-part-2/>

1. For Free fake API for testing and prototyping

Link:- <https://jsonplaceholder.typicode.com/>

1. Install Docker Kong With External Postgresql

Link:-https://jittagornp.github.io/install\_docker\_kong\_with\_external\_postgresql\_on\_ubuntu\_1804.html

Kong Mesh with Docker

Link:- https://docs.konghq.com/mesh/1.3.x/installation/docker/

Step 1:- Download Kong Mesh

docker pull kong/kuma-cp:1.4.0

Step 2:- Run Kong Mesh

Run the control plane with:

docker run \

-p 5681:5681 \

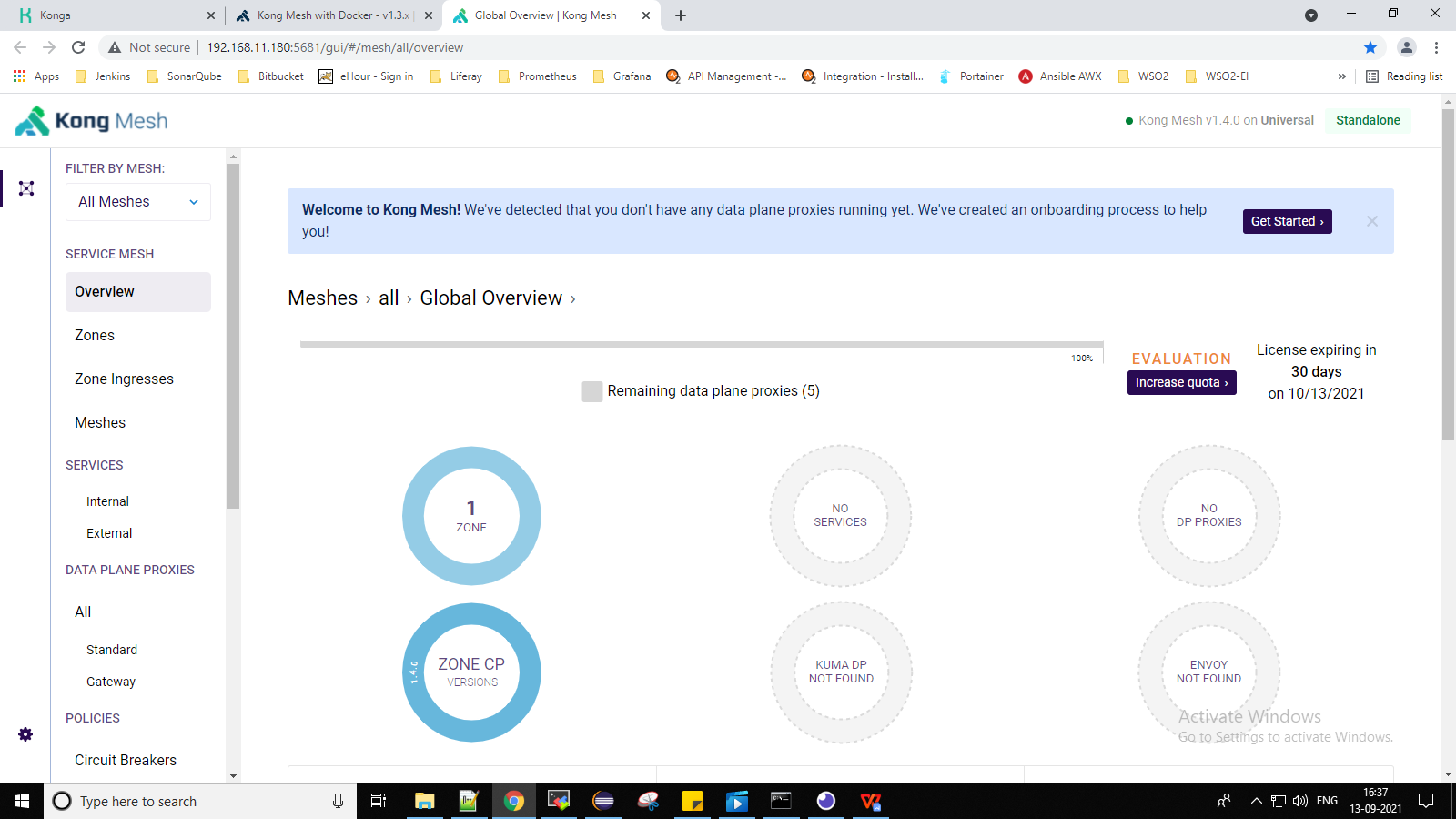
-v /path/to/license.json:/license.json \

-e "KMESH\_LICENSE\_PATH=/license.json" \

kong/kuma-cp:1.4.0 run

Step 3:- Verify the Installation

To access Kong Mesh, navigate to 127.0.0.1:5681/gui OR {host\_ip}:5681/gui to see the GUI.



Kong Map with Docker

Link:- https://github.com/yesinteractive/kong-map/

Step 1:- Export Cluster Configurations to KONG\_CLUSTERS Environment Variable

export KONG\_CLUSTERS='{ "my enterprise cluster": { "kong\_admin\_api\_url": "http://kongapi\_url:8001", "kong\_edit\_config": "true", "kong\_ent": "true", "kong\_ent\_token": "admin", "kong\_ent\_token\_name": "kong-admin-token", "kong\_ent\_manager\_url": "http://kongmanager\_url:8002" }}'

Step 2:- Start Container

docker run -d \

-e "KONGMAP\_CLUSTERS\_JSON=$KONG\_CLUSTERS" \

-e "KONGMAP\_URL=http://url\_to\_kongmap:8100" \

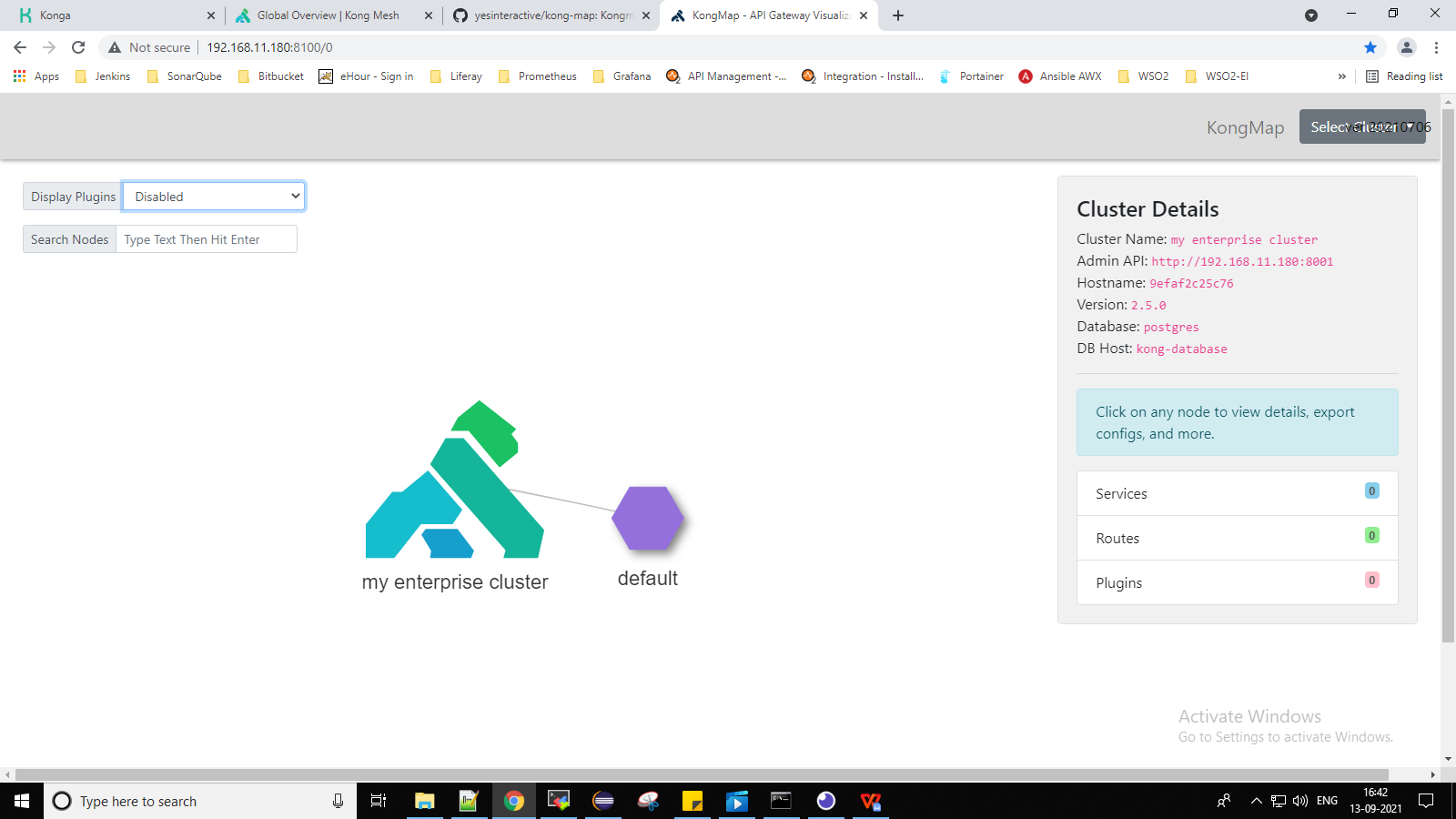
-p 8100:8100 \

-p 8143:8143 \

yesinteractive/kongmap

Step 3:- Verify the Installation

To access Kong Map, navigate to 127.0.0.1:8100 OR {host\_ip}:8100 to see the GUI.



How to Use the Kong Gateway JWT Plugin for Service Authentication

Link:- <https://konghq.com/blog/jwt-kong-gateway>

The Basic Use Case

In this basic use case, I have a login server that accepts login attempts with a user’s email and password. If the email/password checks out, the server generates and signs a JWT and hands it back to the user.

With JWT in hand, the user tries to access our microservice: a simple API server with a single endpoint. Kong Gateway sits in front of your API server, using the JWT plugin for authentication. The user presents his JWT with his request.

First, the plugin verifies the token’s authenticity. Next, it confirms the installation steps of the claims inside the payload. A common claim used is an expiration timestamp for the access token. It’s essentially saying, “This token is valid until this date and time.*”* So, the plugin will check the token’s expiration date.

If the JWT passes all the necessary checks, Kong Gateway grants access to the requested server endpoint. Otherwise, it responds with 401 Unauthorized.

The approach is quite simple:

1. Set up a basic Node.js Express server with a single endpoint.
2. Set up Kong Gateway as an API gateway to your server.
3. Enable the JWT plugin to protect your server endpoint with JWT authentication.

### **1. Set Up a Node.js Express Server and Endpoint**

On your local machine, create a folder for your project. Then, initialize a new Node.js project. In the following examples, I’ll use yarn, but you could use npm too:

|  |  |
| --- | --- |
| 1  2  3 | ~$ mkdir project  ~$ cd project  ~/project$ yarn init  # Use all of the yarn defaults here. |

Next, add Express to your project:

|  |  |
| --- | --- |
| 1 | ~/project$ yarn add express |

In your project folder, create the entry point file, index.js, which will spin up an Express server with a single endpoint. Allow a GET request to /, which will respond with the string, “Hello world!”

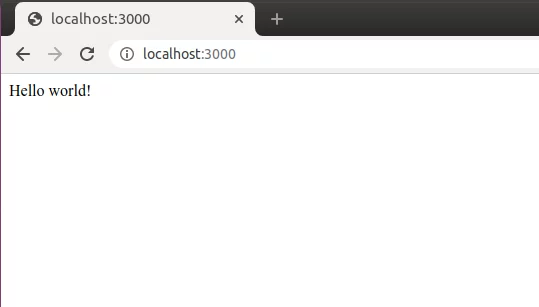
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | /\* PATH: ~/project/index.js  \*/    const express = require('express')  const server = express()  const port = 3000    server.get('/', (req, res) => {    console.log(req.headers)    res.status(200).send('Hello world!')  })    server.listen(port, () => {    console.log(`Server is listening on http://localhost:${port}`)  }) |

That was simple enough! Your single endpoint should log the request headers and then send “Hello world!” back to the client with a 200 status.

Start your server:

|  |  |
| --- | --- |
| 1 | ~/project$ node index.js |

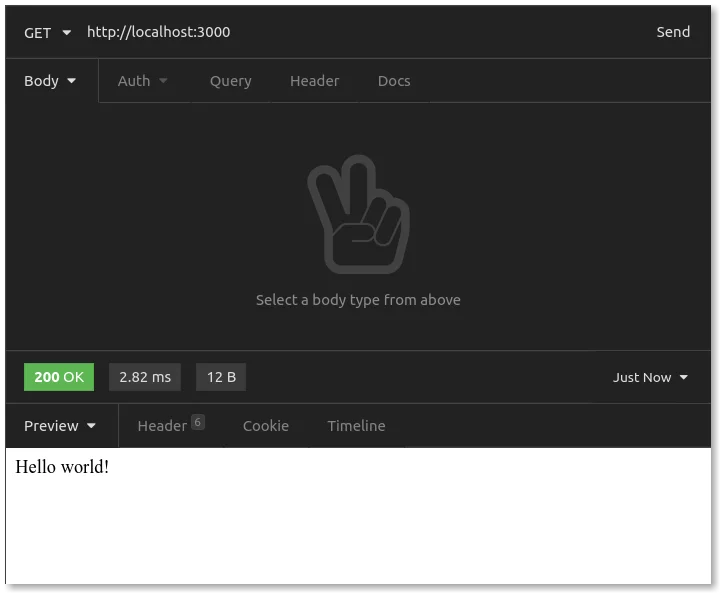
You can use your browser to test this new endpoint by visiting http://localhost:3000.



Your API server endpoint should be working now!

Next, use [Insomnia](https://insomnia.rest/) to send the request and inspect the response. Because of its usability, you’re going to want to use Insomnia exclusively once you start sending requests with a JWT.

In Insomnia, create a GET request to http://localhost:3000. Here’s the response:



In Insomnia, you should get a 200 OK with “Hello world!” in the response body.

It looks like the API server is up and running. Now, it’s time to put Kong Gateway in front of it.

### **2. Set Up Kong Gateway**

I won’t cover the details here, but the [Kong Gateway installation steps](https://konghq.com/blog/set-up-kong-gateway/" \t "https://konghq.com/blog/_blank) may look different depending on your system.

Once you’ve installed Kong, you’ll need to take a few additional steps.

#### ****DB-Less Declarative Configuration****

There are [two primary ways to configure Kong](https://docs.konghq.com/gateway-oss/2.3.x/db-less-and-declarative-config/" \t "https://konghq.com/blog/_blank). **Imperative configuration** issues step-by-step configuration commands to Kong through its admin API. Meanwhile, **declarative configuration** stores the entire configuration in a single .yml file then loads it into Kong upon startup. Additionally, you can configure Kong to hook into your database, providing more control over the different nodes it manages.

For the simple setup example, I’ll use **database-less declarative configuration**. When you start up Kong, you’ll tell it where to find a .yml file with all of the configuration declared within.

In your project folder, run the following command, which generates an initial kong.yml declarative configuration file.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ~/project$ kong config init  ~/project$ tree -L 1  .  ├── index.js  ├── kong.yml  ├── node\_modules  ├── package.json  └── yarn.lock  1 directory, 4 files |

Next, you’ll need to configure the system’s kong.conf file before starting up Kong. If you’re working on Ubuntu, you’ll be working in /etc/kong. Here is a template to copy over and then edit.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | ~/project$ cd /etc/kong  /etc/kong$ sudo su    root:/etc/kong$ tree  .  ├── kong.conf.default  └── kong.logrotate  0 directories, 2 files    root:/etc/kong$ cp kong.conf.default kong.conf |

There are only two edits you need to make in your kong.conf file.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | # PATH: /etc/kong/kong.conf    # Around line 839, uncomment and set to off  database = off    # Around line 1023, uncomment and set an absolute path to kong.yml  declarative\_config = /PATH/TO/YOUR/project/kong.yml |

When Kong starts up, it will be in DB-less mode, meaning it will look to your project’s kong.yml file for a configuration.

Finally, you’ll need to edit your kong.yml  file to set up a gateway in front of your API server “hello world” endpoint.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | /\* PATH: ~/project/kong.yml  \*/    \_format\_version: "2.1"    services:  - name: my-api-server    url: http://localhost:3000/  routes:  - name: api-requests    service: my-api-server    paths:      - /api |

Let’s go over this.

The \_format\_version metadata specifies the version number of your declarative configuration format.

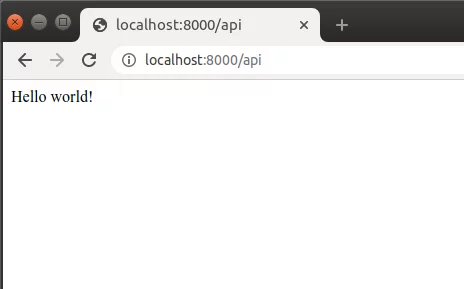
Next, you define your service, which [Kong describes](https://docs.konghq.com/getting-started-guide/2.3.x/expose-services/) as “an entity representing an external upstream API or microservice.” You can name your service my-api-service and specify its URL — you’ll recall that the Express server listens for requests at http://localhost:3000.

Next, define routes, which “determine how (and if) requests are sent to their Services after they reach Kong Gateway.” The (local) URL for Kong is http://localhost:8000. You should declare your route so that Kong listens for requests at http://localhost:8000/api, then routes to your service.

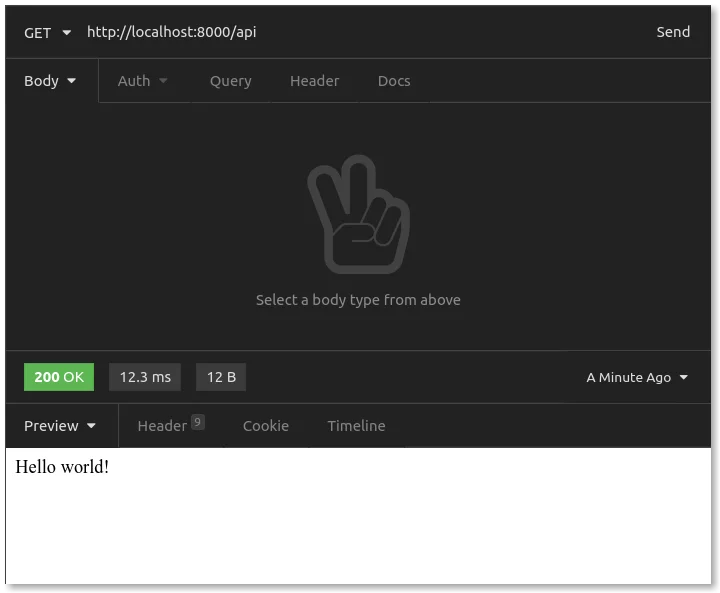
Let’s see this in action. Make sure your Express server is running in a separate terminal. Then, start Kong.

|  |  |
| --- | --- |
| 1 | ~/project$ sudo kong start |

In your browser, go to http://localhost:8000/api:



Similarly, in Insomnia:



Kong Gateway is up. Finally, add authentication.

### **3. Attach JWT Plugin to Kong Gateway**

To add the JWT plugin, add a “plugins” definition to your kong.yml file:

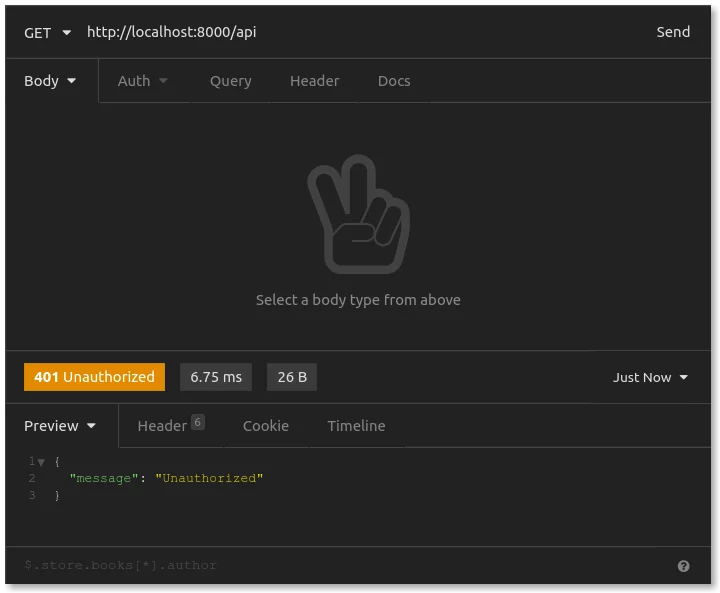
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | /\* PATH: ~/project/kong.yml  \*/    \_format\_version: "2.1"    services:  - name: my-api-server    url: http://localhost:3000/  routes:  - name: api-requests    service: my-api-server    paths:      - /api  plugins:  - name: jwt    service: my-api-server    enabled: true    config:      key\_claim\_name: kid      claims\_to\_verify:      - exp |

Here, you can add the plugin named jwt and attach it to your service called my-api-server. For its [configuration options](https://docs.konghq.com/hub/kong-inc/jwt/" \l "parameters" \t "https://konghq.com/blog/_blank), tell the plugin to check the exp value to verify that the access token has not expired.

At this point, restart Kong and see what happens:

|  |  |
| --- | --- |
| 1 | ~/project$ sudo kong restart |

When you try the request in Insomnia again, here is the response you should get:



The response is 401 Unauthorized. Excellent! Kong now requires a valid JWT for any requests to your API server. Next, you need to tell Kong what constitutes a valid JWT.

In kong.yml, you need to add a consumer and a credential. Kong [describes consumers](https://docs.konghq.com/2.3.x/getting-started/adding-consumers/" \t "https://konghq.com/blog/_blank) as being “associated with individuals using your Service, and can be used for tracking, access management, and more.” In a more elaborate setting, *every* one of your API users could be a consumer. That’s a use case you can read more about towards the end of this article. In this situation, your login server is your consumer. Your login server will be the entity generating JWTs and handing them out. Users who make a request to Kong will be holding a “login server” JWT.

Edit your kong.yml file by adding the “consumers” and “jwt\_secrets” definitions:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | /\* PATH: ~/project/kong.yml  \*/    \_format\_version: "2.1"    services:  - name: my-api-server    url: http://localhost:3000/  routes:  - name: api-requests    service: my-api-server    paths:      - /api  plugins:  - name: jwt    service: my-api-server    enabled: true    config:      key\_claim\_name: kid      claims\_to\_verify:      - exp  consumers:    - username: login\_server\_issuer  jwt\_secrets:    - consumer: login\_server\_issuer      secret: "secret-hash-brown-bear-market-rate-limit" |

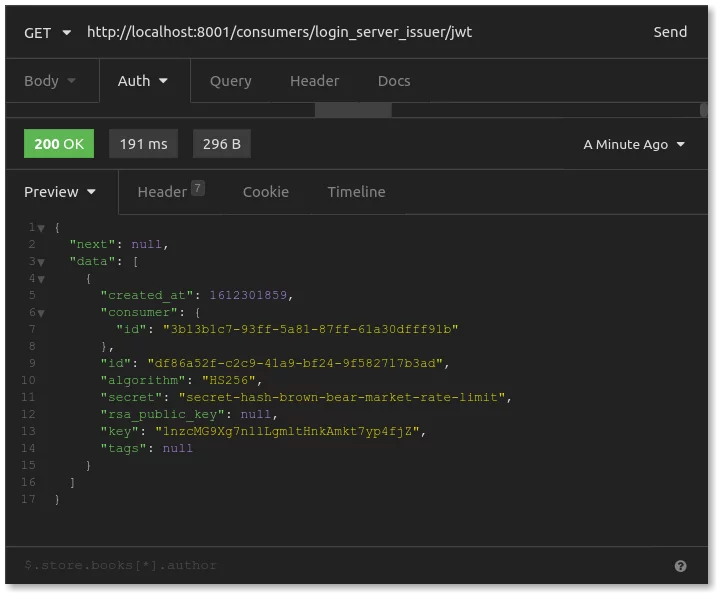
You’ve added a new consumer, named login\_server\_issuer. Then, you added a JWT API gateway credential for that consumer, which contains the secret used to sign JWTs for this consumer. Authentication requires two parts:

1. The kid (key identifier) value in the JWT header, which is a unique identifier that lets the plugin determine *which* consumer allegedly issued this JWT
2. Verification of the consumer’s secret – Was this the secret used to sign this JWT API gateway? If so, then this JWT is authentic.

Before continuing, remember to restart Kong:

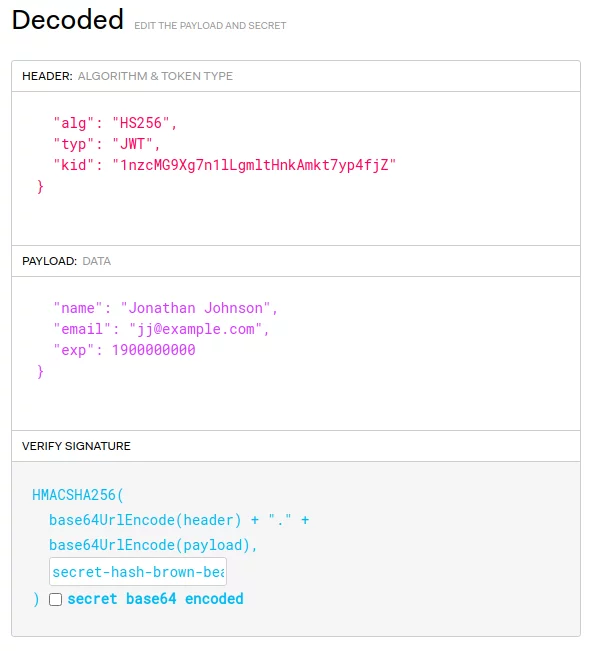
|  |  |
| --- | --- |
| 1 | ~/project$ sudo kong restart |

If you want to generate a JWT for testing, you need the secret (which you have) and the key to use for the kid value. Kong gives us access to that value through its admin API at http://localhost:8001. You send a GET request to the admin API’s endpoint /consumers/CONSUMER-USERNAME/jwt. This gives us information about this consumer’s JWT credential:



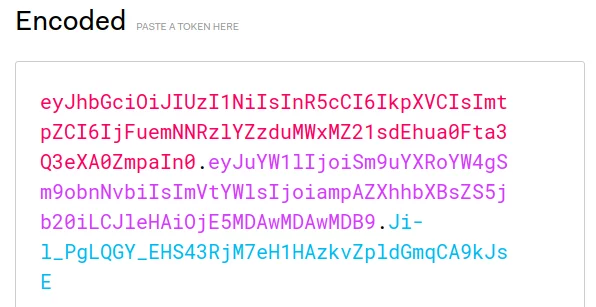
As you inspect this credential’s information, you should see the JWT secret and signing algorithm. What you’re looking for, though, is the key. In the above example, that’s 1nzcMG9Xg7n1lLgmltHnkAmkt7yp4fjZ. **This** is what you use as the kid value in the JWT header. The Kong plugin will see this kid value, track down the associated consumer and secret, then make sure the JWT was signed with that secret.

To test this, let’s start with the happy path. You need a JWT with a header that includes the correct kid value, signed with the right secret. For simplicity, let’s do this at[jwt.io](https://jwt.io/). Here, you can craft your payload, set the signing secret and then copy/paste the resulting JWT.

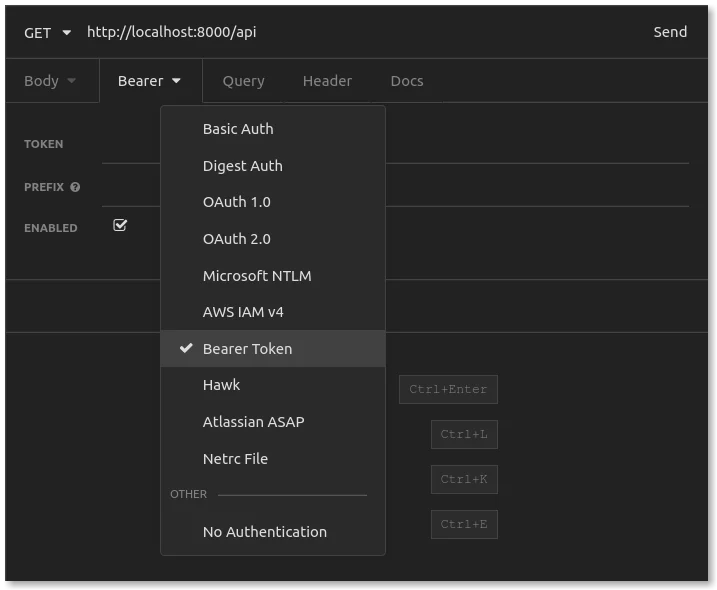


In the payload, the kid must match the key value from above. Also, because you configured the plugin to check JWT token expiration, you should set the exp (Unix timestamp) far into the future. The name and email are inconsequential; they just demonstrate that you can put other helpful data in the JWT payload.

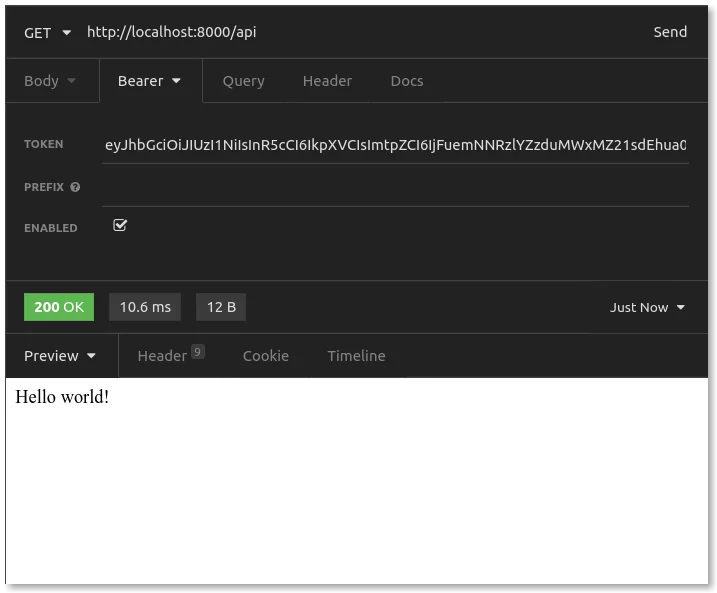
Lastly, include your JWT secret at the bottom for proper signing. The result is an encoded JWT.



Back in Insomnia, you have your original request that resulted in 401. You need to add “Authorization” to that request. Choose “Auth – Bearer Token,” then paste in your encoded JWT from above.

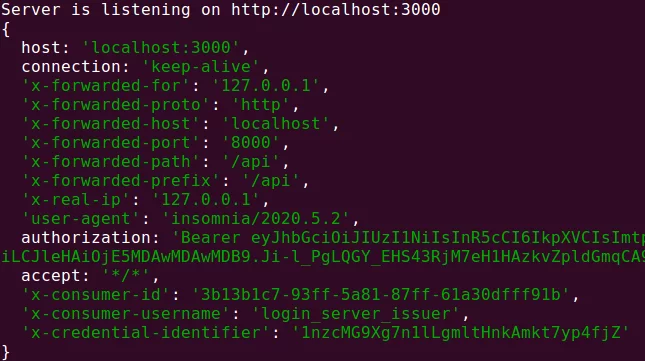


Now, with a valid JWT attached, resend the request.



Your JWT should have been validated, and Kong routed us to the API server!

If you look back at the terminal running the Express server, you’ll recall that you’re logging the request headers to the console. When the JWT plugin authenticates an access token, it [writes some additional values to the upstream headers](https://docs.konghq.com/hub/kong-inc/jwt/" \l "upstream-headers" \t "https://konghq.com/blog/_blank), namely the consumer id, username and credential identifier (the key value).



But what happens if your JWT is not valid? Let’s test and see.

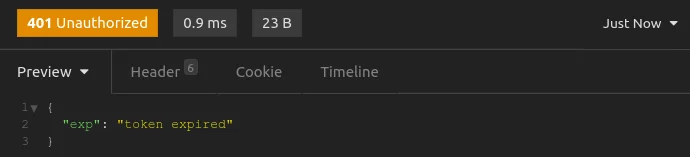
First, sign the JWT with a different secret. Back at [jwt.io](https://jwt.io/" \t "https://konghq.com/blog/_blank), keep the payload, but change the signing secret. Copy the resulting JWT to Insomnia, and send your request again. You’ll get a 401 with “Invalid Signature.”



If your secret is correct, but the kid is incorrect, Kong won’t find an associated credential. Without that credential, there’s no way to find the secret for authenticating the JWT.



Lastly, if the exp value is in the past, then your JWT has expired. Just as you expected, you get the following response.



And that’s it! You should be up and running with Kong Gateway and the JWT Plugin acting as an authentication layer in front of an API server.

# Protecting Services With Kong Gateway Rate Limiting

Link:- <https://konghq.com/blog/kong-gateway-rate-limiting/>

## Create a Node.js Express API Server

To get started, we’ll create a simple API server with a single endpoint that listens for a GET request and responds with “hello world.” At the command line, create a project folder and initialize a Node.js application:

|  |  |
| --- | --- |
| 1  2  3  4 | ~/$ mkdir project  ~/$ cd project  ~/project$ yarn init  // Accept all defaults |

Then, we’ll add Express, which is the only package we’ll need:

|  |  |
| --- | --- |
| 1 | ~/project$ yarn add express |

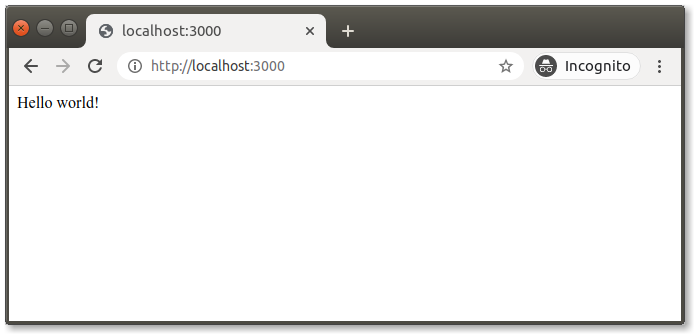
Lastly, let’s create a simple server with our “hello world” endpoint. In your project folder, create an index.js file with these contents:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | /\* PATH: ~/project/index.js  \*/  const express = require('express')  const server = express()  const port = 3000  server.get('/', (req, res) => {  console.log(req.headers)  res.status(200).send('Hello world!')  })  server.listen(port, () => {  console.log(`Server is listening on http://localhost:${port}`)  }) |

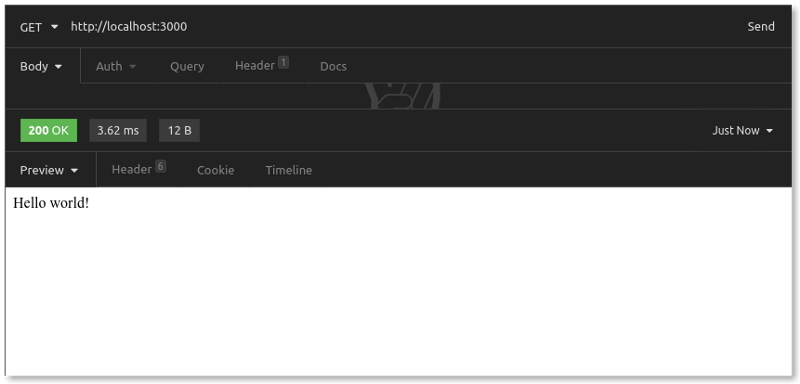
Now, spin up your server:

|  |  |
| --- | --- |
| 1 | ~/project$ node index.js |

In your browser, you can visit http://localhost:3000. Here’s what you should see:



Let’s also use [Insomnia](https://insomnia.rest/" \t "https://konghq.com/blog/kong-gateway-rate-limiting/_blank), a desktop client for API testing. In Insomnia, we send a GET request to http://localhost:3000.



Our Node.js API server with its single endpoint is up and running. We have our 200 OK response.

Keep that terminal window with the node running. We’ll do the rest of our work in a separate terminal window.

## Install and Set Up Kong Gateway

Next, we’ll [install Kong Gateway](https://konghq.com/install/" \t "https://konghq.com/blog/kong-gateway-rate-limiting/_blank) to sit in front of our API server. These steps will vary depending on your local environment.

Simple usage of the Rate Limiting plugin supports configuring Kong in DB-less mode with a declarative configuration. That means, instead of sending configurations to Kong Gateway one step at a time, with those configurations stored in a database, we can use a single declarative .yml file for specifying our entire Kong configuration.

After installing Kong, we generate a starter .yml file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ~/project$ kong config init  ~/project$ tree -L 1  .  ├── index.js  ├── kong.yml  ├── node\_modules  ├── package.json  └── yarn.lock  1 directory, 4 files |

We’ll come back to that kong.yml file in a moment.

Now, let’s tell Kong where to look for that declarative configuration file upon startup. In /etc/kong, there is a kong.conf.default file that we’ll need to copy as kong.conf and then edit:

|  |  |
| --- | --- |
| 1  2  3 | ~/project$ cd /etc/kong  /etc/kong$ sudo su  root:/etc/kong$ cp kong.conf.default kong.conf |

Next, we edit the kong.conf file. You’ll likely need root privileges to do this. There are two edits that we need to make:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | # PATH: /etc/kong/kong.conf    # line ~839: Uncomment this line and set to off  database = off    # line ~1023, Uncomment this line. Set to absolute path to kong.yml  declarative\_config = /PATH/TO/YOUR/project/kong.yml |

## Configure Kong Gateway With Our Service and Routes

Before we start up Kong, let’s edit the declarative configuration file generated in our project folder. It should look like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # PATH: ~/project/kong.yml    \_format\_version: "2.1"    services:  - name: my-api-server    url: http://localhost:3000/    routes:    - name: api-routes      paths:      - /api |

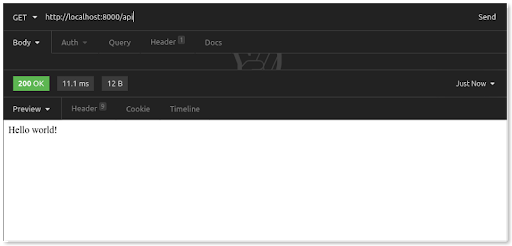
Let’s walk through what this configuration does. After setting the syntax version (2.1), we configure a new upstream service for which Kong will serve as an API gateway. Our service, which we name my-api-server, listens for requests at the URL http://localhost:3000.

We associate a route (arbitrarily named api-routes) for our service with the path /api. Kong Gateway will listen for requests to /api, and then route those requests to our API server at http://localhost:3000.

With our declarative configuration file in place, we start Kong:

|  |  |
| --- | --- |
| 1 | ~/project$ sudo kong start |

Now, in Insomnia, we send a GET request through Kong Gateway, which listens at http://localhost:8000, to the /api path:



Note that, by sending our request to port 8000, we are going *through* Kong Gateway to get to our API server.

## Add the Kong Gateway Rate Limiting Plugin

Now that we have Kong Gateway sitting in front of our API server, we’ll add in the Rate Limiting Plugin and test it out. We will need to add a few lines to our kong.yml declarative configuration file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | # PATH: ~/project/kong.yml    \_format\_version: "2.1"    services:  - name: my-api-server    url: http://localhost:3000/    routes:    - name: api-routes      paths:      - /api    plugins:    - name: rate-limiting      config:        minute: 5        hour: 12        policy: local |

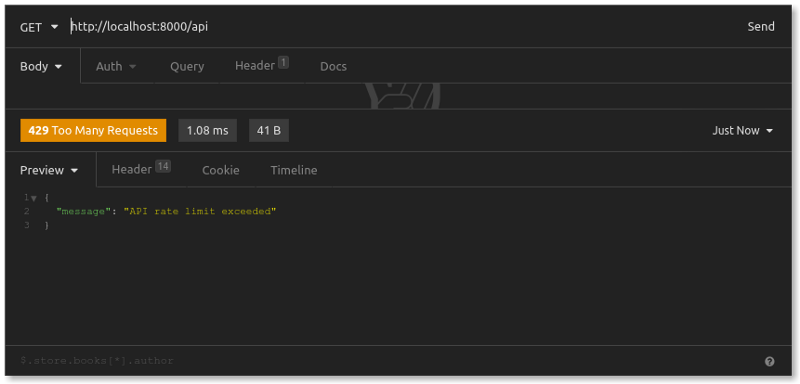
We’ve added the entire plugins section underneath our my-api-server service. We specify the name of the plugin, rate-limiting. This name is *not* arbitrary but refers to the actual rate-limiting plugin in the Kong package.

In this first run, we’ve configured the plugin with minute: 5, which allows for up to five requests per minute. We’ve also added hour : 12, which limits the requests per hour to 12. We’re using the local policy, which has to do with how Kong will store and increment request counts. We’ll talk about this policy configuration more below.

Let’s restart Kong:

|  |  |
| --- | --- |
| 1 | ~/project$ sudo kong restart |

Now, back in Insomnia, we test out sending some requests to our API endpoint. If you send a request every few seconds, you’ll see that the first five requests received a 200 OK response. However, if you exceed five requests within a minute:



The Rate Limiting plugin detects that requests have exceeded the five-per-minute rule. It doesn’t let the subsequent request through to the API server. Instead, we get a 429 Too Many Requests response.

Our plugin works!

If you wait a minute and then try your requests again, you’ll see that you can get another five successful requests until Kong blocks you again with another 429.

At this point, we’ve sent 10 requests over several minutes. But you’ll recall that we configured our plugin with hour: 12. That means our 11th and 12th requests will be successful. However, the system will reject our 13th request  even though we haven’t exceeded the five-per-minute rule . That’s because we’ll have exceeded the 12-per-hour rule.

The Rate Limiting plugin allows you to limit the number of requests per second, minute, hour, day, month and year.

### Rate Limiting “Similar” Requests

We’ve configured Kong to count (and limit) requests to our server in our simple use case so far. More likely, we’ll want to apply our rate limits to *similar* requests. By “similar,” we might mean “coming from the same IP address” or “using the same API key.”

For example, let’s say that all users of our API server need to send requests with a unique API key set in headers as x-api-key. We can configure Kong to apply its rate limits on a per-API-key basis as follows:

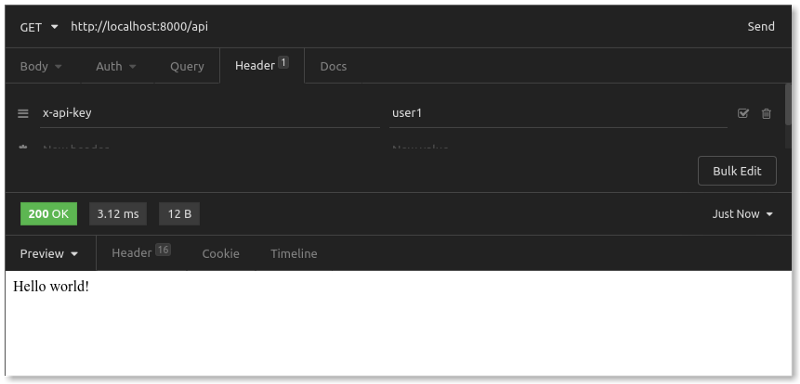
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | # PATH: ~/project/kong.yml    \_format\_version: "2.1"    services:  - name: my-api-server    url: http://localhost:3000/    routes:    - name: api-routes      paths:      - /api    plugins:    - name: rate-limiting      config:        minute: 5        hour: 12        policy: local        limit\_by: header        header\_name: x-api-key |

The Rate Limiting plugin will apply its limits by grouping requests according to a value found in the x-api-key in the header. Requests with the same x-api-key will be considered “similar” requests.

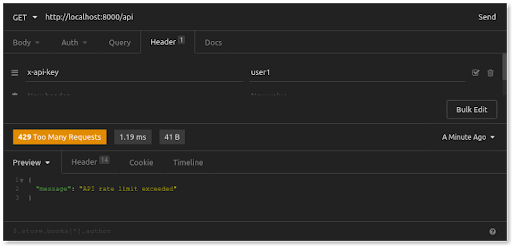
Let’s restart Kong and see this in action:

|  |  |
| --- | --- |
| 1 | ~/project$ sudo kong restart |

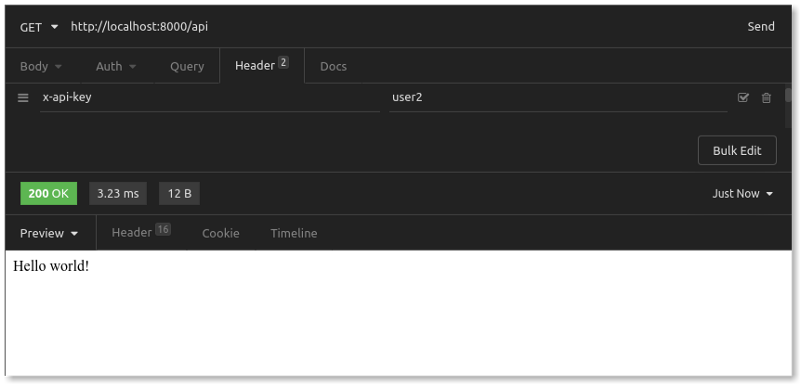
Back in Insomnia, we’ll adjust our GET request slightly by adding a new x-api-key header value. Let’s first set the value to user1. We send the request multiple times. Our first five requests return a 200 OK:



When we exceed the five-per-minute rule, however, this is what we see:



If we change the x-api-key to a different value (for example, user2), we immediately get 200 OK responses:



Requests from user2 are counted separately from requests from user1. Each unique x-api-key value gets (according to our rate limiting rules) up to five requests per minute and up to 12 requests per hour.

You can configure the plugin to limit\_by IP address, a header value, request path or even credentials (like [OAuth2](https://konghq.com/blog/kong-gateway-oauth2/" \t "https://konghq.com/blog/kong-gateway-rate-limiting/_blank) or [JWT](https://konghq.com/blog/jwt-kong-gateway" \t "https://konghq.com/blog/kong-gateway-rate-limiting/_blank)).

* Check docker logs using below path:-

/var/lib/docker/containers/9efaf2c25c76/9efaf2c25c76-json.log

* check number of threads in linux or docker

ps -o nlwp <process\_id>