Grey Matter Service Deployment Training

Unstar(S)

These are instructions on how to deploy a new service into Grey Matter. They assume you already have a Grey Matter deployment ready and accessible as a result of following the quickstart deployment training to create a personal EC2 running Grey Matter with Minikube.

They also assume access to the PEM-encoded x509 quickstart certificates, downloadable from here.

Until now we have only needed them on our local machine, to provide authentication from your browser. For configuring the Grey Matter CLI on the EC2 instance, we will need them on the server as well. SCP the certificates to your server, or alternatively use `wget` to download the zip and unzip it directly on the EC2:

```
wget 'https://docs.google.com/uc?export=download&id=1YEyw5vEHrXhDpGuDk9RH
QcQk5kFk38uz' -O quickstart.zip
```

```
unzip quickstart.zip -d certs/
```

Download the latest Grey Matter CLI

We will be configuring the mesh with the latest Grey Matter CLI, available after authentication from Decipher's Nexus repository. Download the latest available version, and untar the package. At time of writing, the latest version is `v1.1.0`, which will be used in the below example.

The simplest way to get the `greymatter` CLI binary onto your server is to upload it from your local machine after downloading it from Decipher's Nexus repository. For Ubuntu you'll want `greymatter-v1.1.0/greymatter.linux` of course.

Upload this file to the server. Then make it executable, and move it into your path while renaming it to `greymatter`:

```
chmod +x greymatter.linux
sudo mv greymatter.linux /usr/local/bin/greymatter
```

> Note: It is absolutely necessary that the binary be renamed and put into your path for it to work. This is a known issue, and will be fixed in future versions.

Configuring the Grey Matter CLI

Much of what follows involves the Grey Matter CLI. This is a standalone binary, usually configured through environment variables, that interacts with the Grey Matter Control API to configure services within the mesh.

We begin by configuring the CLI to connect to the Control API:

```
# A convenient trick for discovering your EC2's host and port
export HOST=$( curl -s http://169.254.169.254/latest/meta-data/public-ipv
4 )
export PORT=$( sudo minikube service list | grep voyager-edge | grep -oP
':\K(\d+)' )

echo https://$HOST:$PORT

# 1
export GREYMATTER_API_HOST="$HOST:$PORT"
# 2
export GREYMATTER_API_PREFIX='/services/gm-control-api/latest'
# 3
export GREYMATTER_API_SSLCERT="$(pwd)/certs/quickstart.crt"
```

```
# 4
export GREYMATTER_API_SSLKEY="$(pwd)/certs/quickstart.key"
# 5
export GREYMATTER_CONSOLE_LEVEL='debug'
# 6
export GREYMATTER_API_SSL='true'
# 7
export GREYMATTER_API_INSECURE='true'
```

These environment variables setup the CLI to

1. Point to your EC2's exposed edge host and IP.
2. Navigate to the control API's root URL.
3. Setup your user certificate
4. and its associated key.
5. Set a logging level appropriate for this tutorial.
6. Use `https` instead of `http` when making requests to the Control API service.
7. Don't attempt to verify the server identity (for this tutorial).

If you would like this configuration to persist, place it into your `~/.bash_profile`.

Verify your CLI connection with

```
greymatter list cluster
```

If successful, this should return a JSON representation of the "clusters" (meaning, in Envoy parlance, the logical services known to the discovery mechanism).

> NOTE: `cluster` is singular, and a request for `clusters` will fail. This pattern holds with the CLI for most object types.

Launching a new service

The first stage to installing a new service into the mesh is to launch the actual service and its sidecar. For our purposes here, we use our Minikube deployment from a previous tutorial, and therefore the deployment configuration is a Kubernetes configuration file. For simplicity, we launch a common Decipher test image: the Fibonacci service.

The files discussed in this section can be found here in fib.zip. Unzip the `fib/` directory so you can follow along. We will examine each file as we deploy it.

You can download this file directly onto your EC2 instance with `wget`:

```
wget 'https://docs.google.com/uc?export=download&id=10s3emQdJvpLsOa0bJM4W
_u66f4OxVOCY' -O fib.zip
```

```
unzip fib.zip

cd fib/
```

Kubernetes configuration

Here is a Kubernetes configuration typical of a Grey Matter service deployment (`fib/1_kubernetes/fib.yaml` in the zip). It contains two containers, one for the Grey Matter sidecar, and one for the Fibonacci service itself. These both launch into a common pod, labeled “fibonacci”.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fibonacci
spec:
  selector:
    matchLabels:
      app: fibonacci
  replicas: 1
  template:
    metadata:
      labels:
        app: fibonacci
    spec:
      containers:
      - name: fibonacci
        image: docker.production.deciphernow.com/services/fibonacci:lates
t
        ports:
        - containerPort: 8080
      - name: sidecar
        image: docker.production.deciphernow.com/deciphernow/gm-proxy:lat
est
        imagePullPolicy: Always
        ports:
        - name: proxy
          containerPort: 9080
        - name: metrics
          containerPort: 8081
        env:
        - name: PROXY_DYNAMIC
          value: "true"
        - name: XDS_CLUSTER
          value: fibonacci
        - name: XDS_HOST
          value: control.default.svc.cluster.local
```

```
        - name: XDS_PORT
          value: "50000"
      imagePullSecrets:
      - name: docker.secret
```

> Note: This is a minimal deployment configuration. In case you're unfamiliar with Kubernetes configuration: The header boilerplate and the names of the configured `containerPorts` may seem particularly ripe for pruning, but resist the temptation. We will revisit this and explain why in the mesh configuration training.

Notice that the service itself is listening on port 8080, and the sidecar is listening on 9080 and 8081. Soon we will configure the sidecar to proxy 9080 → 8080, and expose collected statistics to Prometheus on 8081.

Another salient feature is that the sidecar is largely unconfigured. It is given only enough information to know that it should contact the control plane to request configuration (`PROXY_DYNAMIC`), where to do that (`XDS_HOST` and `XDS_PORT`), and what configuration to request (that of `XDS_CLUSTER`).

> Note: If you are deploying your service into a non-default namespace, `control.default.svc.cluster.local` should instead read `control.YOURNAMESPACE.svc.cluster.local`.

Apply this configuration with

```
sudo kubectl apply -f 1_kubernetes/fib.yaml
```

Your new deployment should appear in the `get pods` listing shortly, with 2/2 pods ready.

```
ubuntu@ip-172-31-40-178:~$ sudo kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
catalog-57fb9776d7-vkslz            2/2       Running   0          20h
control-7b7cc7645f-c5b68            1/1       Running   0          20h
dashboard-9cbfb56fc-dxmsc           2/2       Running   0          20h
data-694667746b-jkxh2               2/2       Running   0          20h
data-internal-567cf95cf-kh7ct       2/2       Running   0          20h
data-mongo-0                        1/1       Running   0          20h
edge-9bfbb8f86-thgx2                1/1       Running   0          20h
fibonacci-8589f97b88-pcx9d          2/2       Running   0          19h
gm-control-api-794f47cb84-hrw52     2/2       Running   0          20h
internal-data-mongo-0               1/1       Running   0          20h
internal-jwt-security-99859778d-m6qz9   2/2       Running   3          20h
internal-redis-564b47c857-2d2sb     1/1       Running   0          20h
jwt-security-6dc55c59bc-89m5b       2/2       Running   3          20h
postgres-slo-0                      1/1       Running   0          20h
prometheus-55b9dbb8c6-jkstx         2/2       Running   0          20h
redis-57f7659bc7-knwhl              1/1       Running   0          20h
slo-658df848f5-cfhns                2/2       Running   0          20h
voyager-edge-869c87c588-t6zjz       1/1       Running   0          20h
ubuntu@ip-172-31-40-178:~$
```

Grey Matter sidecar configuration

At this point, the service and the sidecar are both launched with ports open, but the sidecar is largely unconfigured. It only knows to keep checking in with the Grey Matter control plane for updated configuration. We will now send that configuration for the fibonacci service to the control plane.

There are six configuration objects necessary to configure the sidecar, represented by six JSON files (in `fib/2_sidecar/` in the zip). We will go through each of them as we deploy them.

> Note: A successful response to each `greymatter` CLI request will closely resemble the object sent, with three caveats:
>
> 1. The response omits misspelled key names, which can be a source of confusion, so check carefully that all keys sent were received as expected.
> 2. It may omit settings that match the defaults, depending on the command. E.g., `edit` omits defaults.
> 3. It contains a checksum, which when using `edit` must not be changed (because it's used by Grey Matter Control to enforce isolation of updates).

Cluster

In Envoy parlance, a "cluster" corresponds to a virtual service load balanced across one or more cluster "instances". So for example, a hypothetical bookmarks service would be considered a "cluster" to Grey Matter, and the actual running code, along with its sidecar, would make up an instance of this bookmarks cluster. We create a cluster *configuration* whenever we want a Grey Matter sidecar proxy to be able to communicate with instances of another cluster, or with the

service itself. In other words, anything at all the sidecar will need to communicate with is a “cluster”.

Here we configure just one cluster, corresponding to the actual service code running next to the sidecar. We will later refer to this cluster (in our `proxy` config) using the `cluster_key` specified here to setup the request proxying.

Here is the contents of `cluster.json`:

```
{
    "zone_key": "zone-default-zone",
    "cluster_key": "fibonacci-service",
    "name": "service",
    "instances": [
        {
            "host": "localhost",
            "port": 8080
        }
    ]
}
```

We send it to Grey Matter with

```
greymatter create cluster < 2_sidecar/cluster.json
```

Domain

Domains provide namespaces for routes. To simplify and decentralize routing configuration, we often recommend making a new domain for each service.

Here is the contents of `domain.json`, a minimal configuration to create the ‘fibonacci’ domain. (For our purposes here, we create a whole new domain for each service):

```
{
    "zone_key": "zone-default-zone",
    "domain_key": "fibonacci",
    "name": "*",
    "port": 9080
}
```

We send it to Grey Matter with

```
greymatter create domain < 2_sidecar/domain.json
```

Listener

Listener configuration connects a domain, IP, and port to a protocol.

> Note: It is a potential source of confusion that domains and listeners overlap in functionality, and they may in fact be merged in future versions. Listeners and domains differ mainly in the one-many relationship between listeners and their associated domains, i.e., you can configure a single listener and attach it to multiple domains. The configuration associated with a listener would then apply to multiple domains without duplication.

Here is the contents of `listener.json`:

```
{
    "zone_key": "zone-default-zone",
    "listener_key": "fibonacci-listener",
    "domain_keys": ["fibonacci"],
    "name": "fibonacci",
    "ip": "0.0.0.0",
    "port": 9080,
    "protocol": "http_auto"
}
```

We send it to Grey Matter with

```
greymatter create listener < 2_sidecar/listener.json
```

Shared Rules

Shared rules contain reusable traffic management configuration, for implementing traffic shaping, fault-injection, A/B testing, canary deployments, dark launches, shadowing, splitting, etc. Routes refer to shared rules by their `shared_rules_key`, as will be demonstrated next.

The following configuration object implements a minimal shared rule for simple traffic routing to the cluster we defined above, using the “light” (normal, live traffic path, along which a response will be returned). Other possibilities might involve the “dark”, (a target for a send-and-forget copy of the request) or “tap” (a target for a copy of the request, where the response is compared to that of the “light” path) paths.

Here is the contents of `shared_rules.json`:

```
{
    "zone_key": "zone-default-zone",
    "shared_rules_key": "fibonacci-shared-rules",
    "name": "fibonacci",
    "default": {
        "light": [
            {
                "cluster_key": "fibonacci-service",
```

```
                "weight": 1
            }
        ]
    }
}
```

We send it to Grey Matter with

```
greymatter create shared_rules < 2_sidecar/shared_rules.json
```

Route

Routes connect URL paths to domains, which are later connected to clusters in the `proxy` configuration. Notice that this configuration object references the `shared_rules_key` defined above.

Here is the contents of `route.json`:

```
{
    "zone_key": "zone-default-zone",
    "domain_key": "fibonacci",
    "route_key": "fibonacci-route",
    "path": "/",
    "shared_rules_key": "fibonacci-shared-rules"
}
```

We send it to Grey Matter with

```
greymatter create route < 2_sidecar/route.json
```

Proxy

Finally, a proxy configuration ties all of the configuration we've been building up so far to an *actual sidecar*. You may recall earlier we configured the sidecar via environment variable to request configuration for a particular cluster name? That value (`XDS_CLUSTER`) must match the `name` key in this configuration file, which associates it with domains and listeners.

This configuration also configures two of the most important Grey Matter filters, for metrics and observables.

Here is the contents of `proxy.json`:

```
{
    "zone_key": "zone-default-zone",
    "proxy_key": "fibonacci-proxy",
```

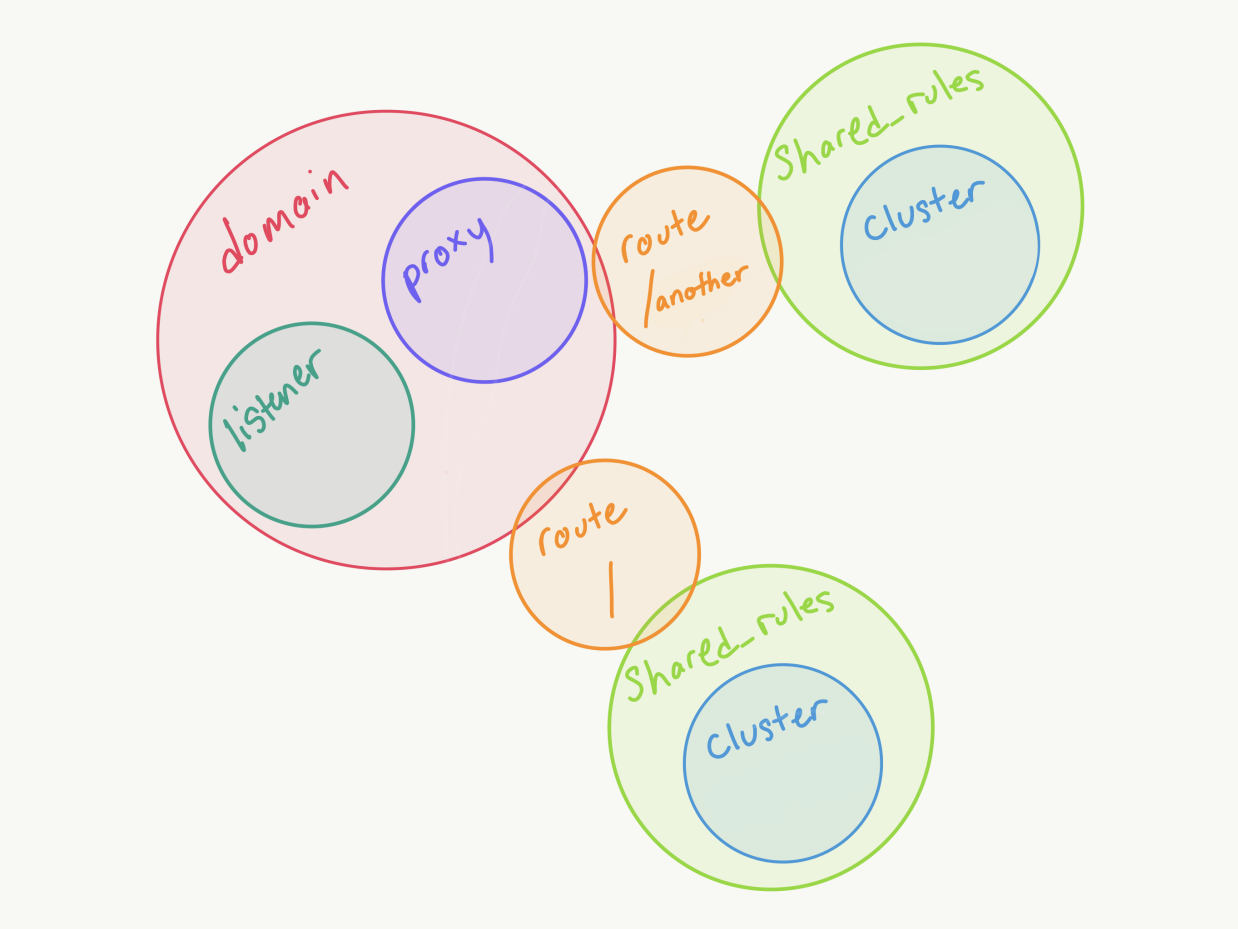
```
    "domain_keys": ["fibonacci"],
    "listener_keys": ["fibonacci-listener"],
    "name": "fibonacci",
    "active_proxy_filters": ["gm.metrics", "gm.observables"],
    "proxy_filters": {
        "gm_metrics": {
            "metrics_port": 8081,
            "metrics_host": "0.0.0.0",
            "metrics_dashboard_uri_path": "/metrics",
            "metrics_prometheus_uri_path": "/prometheus",
            "metrics_ring_buffer_size": 4096,
            "prometheus_system_metrics_interval_seconds": 15,
            "metrics_key_function": "depth"
        },
        "gm_observables": {
            "emitFullResponse": false,
            "useKafka": false,
            "eventTopic": "observables",
            "enforceAudit": false,
            "topic": "fibonacci",
            "kafkaZKDiscover": false,
            "kafkaServerConnection": "kafka-default.fabric.svc:9092"
        }
    }
}
```

We send it to Grey Matter with

```
greymatter create proxy < 2_sidecar/proxy.json
```

What?

It would be understandable if, by this point, you were lost in the sea of configuration files. Logical separation enables minute control of each conceptual aspect of the system, but it does make it more difficult to comprehend at a glance. Let's review at a high level to cement the big picture.



> Note: All configuration above uses the same zone_key, which provides a logical scope to the configuration. Zones represent logical mesh deployments, and may be used to support multi-tenancy. This particular zone key was setup for us when the helm charts for Grey Matter were created. You can list the configured zones with `greymatter list zone`.
>
> `TODO`: Write-up overview script.

Grey Matter edge configuration

We've now configured the sidecar itself, but there's one other instance of Grey Matter proxy in the request chain that we haven't yet configured: the edge proxy. It runs exactly the same code as a sidecar, but with different configuration that causes it to behave as a reverse proxy for the whole deployment (all other sidecars).

The form of its configuration will be familiar, though with slightly more complex route configuration. It should give you some perspective on the variation in configuration possible with Grey Matter proxy.

There are four configuration objects necessary to configure the edge for our new service, represented by four JSON files (in `fib/3_edge/` in the zip). We will go through each of them as we deploy them.

> Note: There are four configuration files *in addition to* the configuration the edge proxy already has. It was configured as an edge proxy by the Helm installation, and what we're doing here is *adding* some configuration so that the edge proxy will be able to route traffic to the Fibonacci service.

Edge Cluster

Here is the contents of `fib-cluster.json`:

```
{
    "zone_key": "zone-default-zone",
    "cluster_key": "edge-fibonacci-cluster",
    "name": "fibonacci",
    "instances": [],
}
```

We send it to Grey Matter with

```
greymatter create cluster < 3_edge/fib-cluster.json
```

Edge Shared Rules

Here is the contents of `fib-shared_rules.json`:

```
{
    "zone_key": "zone-default-zone",
    "shared_rules_key": "edge-fibonacci-shared-rules",
    "name": "fibonacci",
    "default": {
        "light": [
            {
                "cluster_key": "edge-fibonacci-cluster",
                "weight": 1
            }
        ]
    }
}
```

We send it to Grey Matter with

```
greymatter create shared_rules < 3_edge/fib-shared_rules.json
```

Edge Route (for route with trailing slash)

Here is the contents of `fib-route.json`:

```
{
    "zone_key": "zone-default-zone",
    "domain_key": "edge",
    "route_key": "edge-fibonacci-route-slash",
    "path": "/services/fibonacci/1.0/",
    "prefix_rewrite": "/",
    "shared_rules_key": "edge-fibonacci-shared-rules",
}
```

We send it to Grey Matter with

```
greymatter create route < 3_edge/fib-route.json
```

Edge Route 2 (adds missing trailing slash)

Here is the contents of `fib-route-2.json`:

```
{
    "zone_key": "zone-default-zone",
    "domain_key": "edge",
    "route_key": "edge-fibonacci-route",
    "path": "/services/fibonacci/1.0",
    "prefix_rewrite": "/services/fibonacci/1.0/",
    "shared_rules_key": "edge-fibonacci-shared-rules",
}
```

We send it to Grey Matter with

```
greymatter create route < 3_edge/fib-route-2.json
```

If you've followed alone up to this point, you should have a service deployed! Try to reach it at `https://{your-ec2-public-ip}:{port}/services/fibonacci/1.0/`. If you get "Alive", try requesting the 37th Fibonacci number with `/services/fibonacci/1.0/fibonacci/37`. If you get `no healthy upstream` instead, try waiting for a few minutes before retrying. The changes to the control plane make take a few minutes to propogate changes down to the sidecars.

See the Troubleshooting section below if for any other reason your service seems unresponsive.

Catalog service configuration

If your service responds to your call, there is yet one final step remaining: Your Catalog service entry. The Grey Matter Intel 360 Dashboard depends on the Catalog service for information on each running service, and we will make an entry for our service now by POSTing to the Catalog service's `/clusters` endpoint.

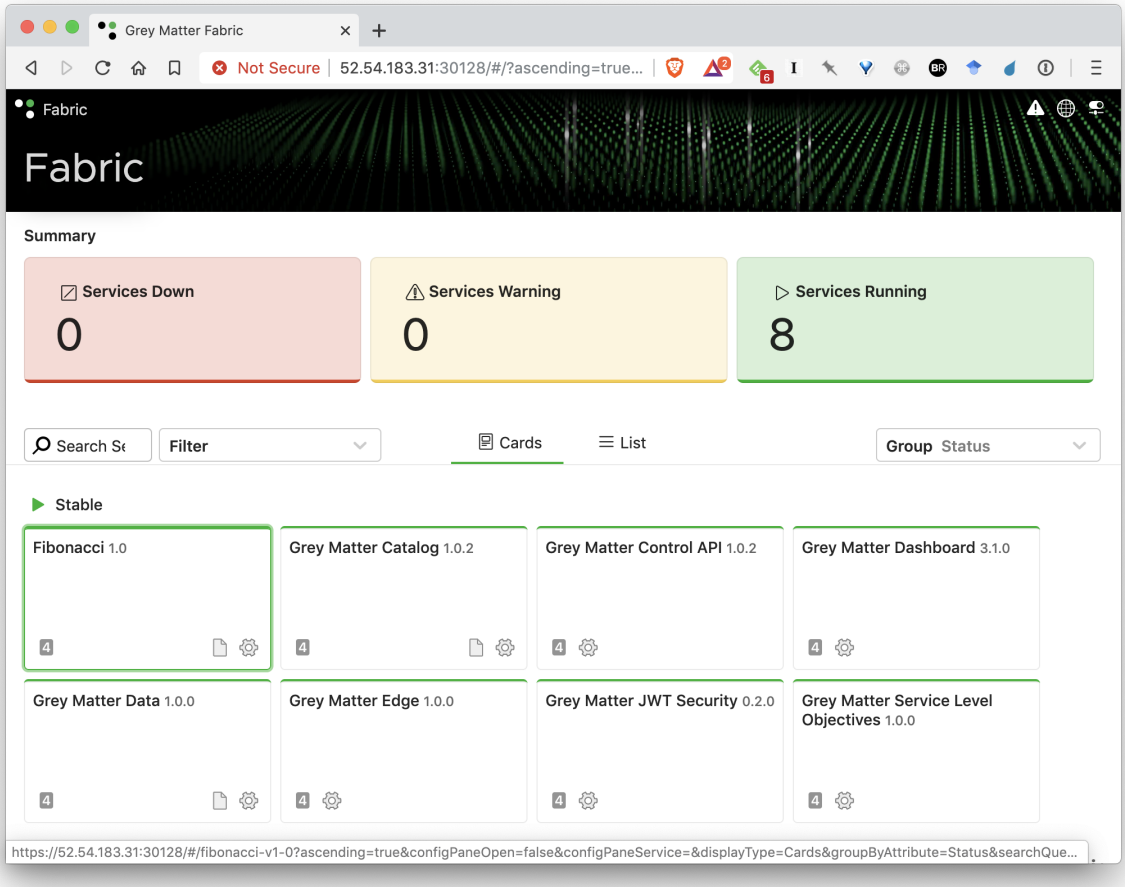
In the same terminal where you set the environment variables for the Grey Matter CLI, in the `fib/` directory, run this to create an entry in the Catalog service. This POSTs the contents of

```
{
    "clusterName": "fibonacci",
    "zoneName": "zone-default-zone",
    "name": "Fibonacci",
    "version": "1.0",
    "owner": "Decipher",
    "capability": "Tutorial",
    "runtime": "GO",
    "documentation": "/services/fibonacci/1.0/",
    "prometheusJob": "fibonacci",
    "minInstances": 1,
    "maxInstances": 2,
    "authorized": true,
    "enableInstanceMetrics": true,
    "enableHistoricalMetrics": true,
    "metricsPort": 8081
'
```

Send it to the Catalog service with

```
curl -XPOST https://$GREYMATTER_API_HOST/services/catalog/latest/clusters
 --cert $GREYMATTER_API_SSLCERT --key $GREYMATTER_API_SSLKEY -k -d "@4_ca
talog/entry.json"
```

Congratulations, you have successfully deployed a service to Grey Matter!



Troubleshooting

> `TODO` : Expand and merge with installation troubleshooting

If you have any trouble following these instructions, or any feedback, please submit a ticket at https://support.deciphernow.com, where we also maintain a knowledgebase and further troubleshooting information.

Below are a few common problems encountered with these instructions in particular and associated explanations and solutions.

404

If the 404 is only for the correct URL for your service (`/services/fibonacci/1.0/`) then a missing route configuration is the most likely cause. Check that the response from control when you did your `greymatter create route < 2_sidecar/route.json` looks correct.

“No healthy upstream”

If you just deployed your service, wait a few minutes. To avoid network chattiness, the control plane sometimes waits between updates. If it has been more than five minutes and the error is still there, the sidecar is likely not correctly configured to talk to its service.

“Upstream connect error or disconnect/reset before headers”

The edge proxy is likely not configured correctly to talk to the service, or the service cannot be found for some other reason. Check the service announcement/discovery pathway.

1. In the Kubernetes config, is the service’s `spec.template.metadata.labels` `app: fibonacci` label intact?
2.