SingularityNet Developer Workshop - Singapore

Summary	In this codelab, you will interact and see the various tools to enable you to build, share and monetize AI services at scale. Specifically you would wrap, deploy and interact with a service you would deploy to singularitynet platforms
URL	-
Category	DEV
Environment	python, go, protobuf, google's grpc, snet-cli, snet-daemon
Status	-
Feedback Link	-
Author	Tesfa Yohannes
Author LDAP	-
Analytics Account	-

Requirements for today

- Create an Etherum wallet: https://www.myetherwallet.com/: Do anyone have issues with

this? Make sure that connect it with metamask.
- Install Docker for linux, Docker for Mac or Docker for Windows: https://docs.docker.com/install/
- Do you have a python interpreter? No, then you might want to install: https://docs.conda.io/en/latest/miniconda.html
- Have a github account.(https://github.com)

- There are other things that we do require, but we will reach them together.

Introduction to the SingularityNet Platform

The singularityNet lets anyone crate, share, and monetize AI services at scale. It is the world's decentralized AI network. We gathered the leading minds in machine learning and blockchain to democratize access to AI technology. Now anyone can take advantage of a global network of AI algorithms, services, and agents.

- AI Decentralized
- AI Democratized

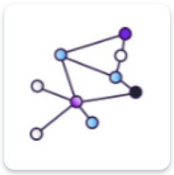
SingularityNet Foundation

Decentralized platform
for AI agents

powered by AGI utility
token

perform services for external
customers or for other AI agents

monetize
AI creations



A society and economy
of AIs

Network governance
and economic logic

Customers can pay in USD, Euro, yen,
won or other means

The SingularityNet Platforms

The easiest way to start interacting with the system is using the first of this tools: The Decentralized AI Marketplace. It's the application store for a **curated** set of services.

The Decentralized AI Marketplace(DApp)

Here is a video that shows the marketplace:
https://www.youtube.com/watch?v=J19yJy-v7Ls

Or we can go the site ourselves:
https://beta.singularitynet.io

The SingularityNet Command Line Tool(CLI)

The SingularityNet DAEMON

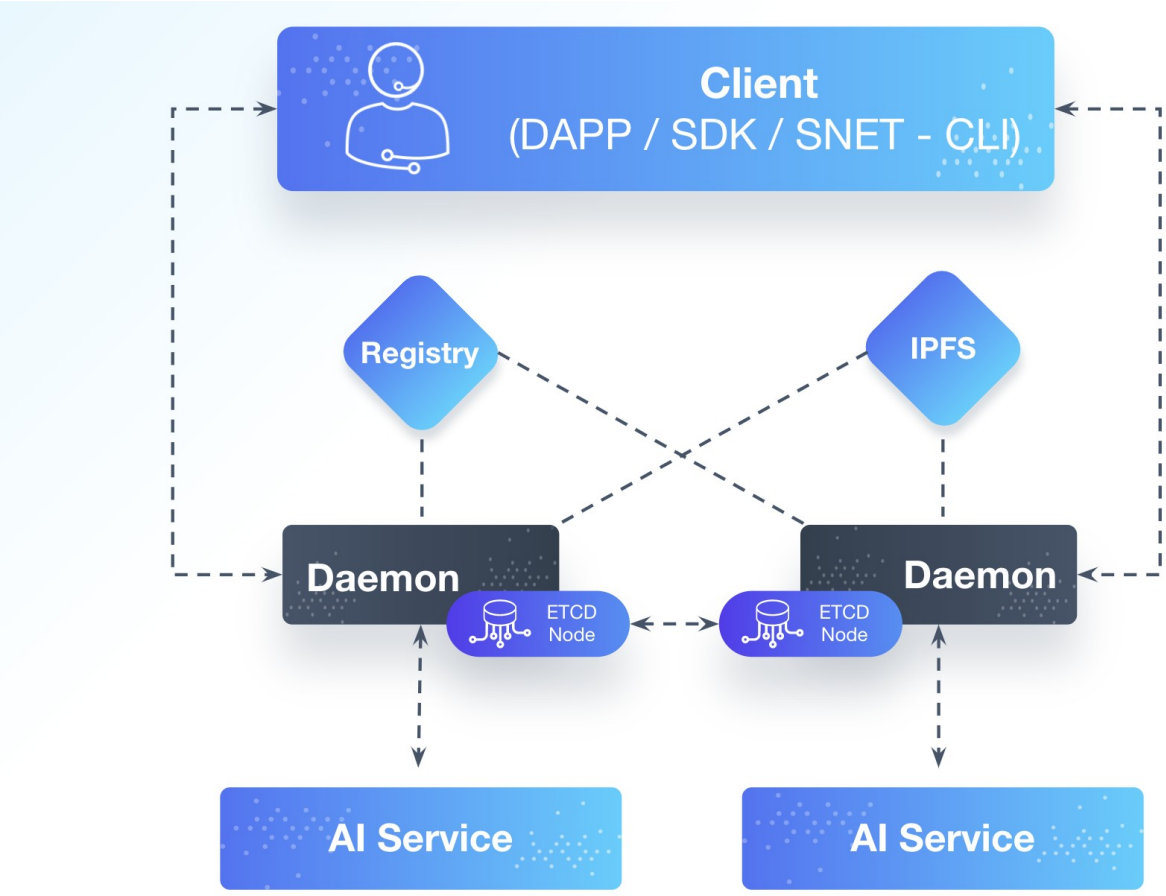
The engine that is going to be connected with the client code that enable the AI service interact with the blockchain to enable authentication, fund reserve and scaling of the service.

The SNET DAEMON is the tool that resides between the Blockchain and the service's endpoint. It is responsible to validate all requests that come from users and to pass them (if everything is ok) to the service.

Once the service process the request and give the DAEMON back a response, the DAEMON delivers it to the user.

The DAEMON interacts with the Registry to get the service metadata and with the IPFS to get the Protobuf API.

DAEMON also uses the ETCD to store all signed payments from users requests.



The SingularityNet DevPortal

This is your resources to give you a more detailed overview of the platform. You can see the hosted version of the application at: https://dev.singularitynet.io/

or run it locally at by cloning it from here:

https://github.com/singnet/dev-portal

You would need to install jekyll and you can get it up and running with the following commands.

```
gem install --user-install jekyll
gem install --user-install jekyll-paginate
gem install --user-install jekyll-sitemap
gem install --user-install jemoji
```

It gives a step by step tutorials to buildng a service in the numerous languages and how to connect it with the various tools available.

Keywords

There are a couple of words that we would mention numerous times during this presentation. Its better if we get them out of the way now.

Service

The AI service itself. It can be just about any program. It only needs to have

- **GRPC** based approach to define input and output.
- Process type approach - we will not cover this today: But can be found at https://dev.singularitynet.io/tutorials/process/. It can be simplified as reading from the standard input/output.
- JSONRPC - a remnant of out alpha days, we can still use JSONRPC to create services. It is one of the primary ways people do interact with ethereum based platforms also. It has its own merits.

Service Description

This is in programming terms the input and output relationship that exists. What is the input format data that we expect? And what is the structure of the response.

Example:
If we do emotion recognition what do we expect?

- Image

What do we return?

- Faces detected and their emotion

But how do we represent this programmatically?

For this project we use google protobufs. Here is an example:

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```

In this dev-workshop we will focus on the GRPC based approach.

Platform

The infrastructure that manages the service. In terms of:

- Payment Mechanism(Blockchain)
- Rating of the service
- The Scale of the service
- The Marketplace API
- etc

We will touch upon this configuration as it pertains to the AI application we are developing.

Testnet

We will be working with simulation of the etherum platform and not the mainnet. In simple terms:

- The Mainnet is where real montary transactions are mined and written to the blockchain
- The testnet is a simulation framework to enable developers work on top of while developing their application. Transactions in this layer are meaningless.

The SingularityNet Platform Contracts

(https://github.com/singnet/platform-contracts)

The Token

ERC-20 is a technical standard used for smart contracts on the Ethereum blockchain for implementing tokens. We have our own token called AGI Token.

The Registry

The smart contract that has information about the token, the service and various assorted information to make a viable call to the endpoint. Namely:

- Name and Description
- IPFS hash to Protobuf file (API)
- Endpoints
- Price (AGI)
- Payment Address.

The Multiparty Escrow

The Multi-Party Escrow smart contract (“MPE”), coupled with our **atomic** *unidirectional* payment channels, enables scalable payments in the platform by minimizing the number of on-blockchain transactions needed between clients and AI service owners.

Note: You can read about this topics and the overall vision in our whitepaper: https://public.singularitynet.io/whitepaper.pdf

Other Parts of the infrastructure we won’t cover here:

- The Reputation System - How do we make sure the services that are in the marketplace are as reputable as possible based on mechanisms that avoid deceit. (link)
- The DAIA - a collection of distributed AI companies working together to bring together a robust distributed AI to the market. (link)

Today’s Service

We will integrate:

- Language Detection from the PolyGlot library.
- Others might want to use example-service repo

As an aside:
- Be sure the service/models that you are using are allowed in commercial setting.

Installation Steps to the library: Natively

Clone the repository:

```
git clone https://github.com/singnet/nlp-services-misc.git nlp-repo
```

Then/Or you can download the file using the following means
Requirement file for python

Then:

```
# in the above folder, change folder to nlp-repo/language-detector
python3.6 -m pip install -r requirement.txt
```

Installation of Steps using Docker

Dockerfile
Then:

```
# in the above folder, change folder to nlp-repo/language-detector
docker build -t singularitynet/language-detection-service .
```

One can get into a docker container with the following command

Check the cheatsheet for docker
https://www.docker.com/sites/default/files/Docker_CheatSheet_08.09.2016_0.pdf

Or better yet: Use the docker documentation
https://docs.docker.com/engine/

If you have the downloaded image:

```
docker load -i workshop_image.tar
```

To start the image:

```
docker run -it -d -p 8005:8005 --name lang singularitynet/language-
detection-service python3.6 start_service.py
```

To create a new terminal. You can do as much as you like until you close the above command

```
docker exec -it lang /bin/bash
```

A final deployed service can be found here:

- Language Detection - https://github.com/singnet/nlp-services-misc
- Example Service - https://github.com/singnet/example-service/issues

What the service itself will constitute of is rather a simple one, here is the code for that:

Code Snippet

```
import sys
import argparse
from polyglot.detect import Detector
from polyglot.text import Text


class LanguageDetector:
    def __init__(self):
        self.result_dict = {}
        self.sentence_dict = {}

    def language_id(self, text):
        sentences = Text(text)
        for sentence in sentences.sentences:
            self.sentence_dict = {}
            for language in Detector(str(sentence)).languages:
                self.sentence_dict[language.name] =
language.confidence
            self.result_dict[str(sentence)] = self.sentence_dict
```

```
        return self.result_dict


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--text', '-t', help='Enter the sentence you
want to detect')
    args = parser.parse_args()
    if args.text == None:
        print('Please enter a sentence. Exiting.')
        sys.exit(0)
    print(args.text)

    lang = LanguageDetector()
    lang.language_id(args.text)
```

It’s time

Simple. We create a class, we pass the input text to a tokenizer that would break the input to separate sentences using its default processing engine and then queries the Detector for the particular language. It returns a dictionary of results. Here is a sample query and a sample result.

```
"Ich bin das Singularität."
"I am the singularity. And I hope to find the best of the world. If
not, as they say in french, Bye"
"እኔ የነጠላነት ደረጃ ነኝ::"
"Ich bin das Singularität. I am the singularity."
"Ich bin das Singularität. I am the singularity. እኔ የነጠላነት ደረጃ ነኝ:: "
```

As an aside: Let’s see examples of AI applications wrapped up in as services in singularitynet.

- MOZI.ai
- Style Transfer
- Emotion Recognition

Organizing code to enable Integration

For now that we have this AI application that we want to wrap up as a service into the singularitynet.

Let's start for real

Create an API for the model using protobuf

What is Protobuf?

- Is messaging protocol that has been developed by Google to enable seamless transfer of data from one endpoint to another.

Historical Aside: We had during our Alpha using JSON rpc to define our endpoints.

This leads to various advantageous.

- We can use this as an API that we would build robust applications with.
- Allows us to create a unique signature for the service that are deployed.

What is the protobuf for the service we have created:

Later change this to code snippet using different markdown syntax or what not:

Protofile

```
syntax = "proto3";
// The input isn't going to be quite complicated. Takes a sentence
message Input {
    string input = 1;
}
// Output a bit complicated. We need the sentence to take
message Output {
    repeated Language language = 1;
}

message Language {
    string sentence = 1;
    repeated Prediction prediction = 2;
}
```

```
message Prediction {
    string language = 2;
    float confidence = 3;
}
// The only service we want to get
service LanguageDetect {
    rpc infer (Input) returns (Output) {
    }
}
```

Create a GRPC endpoint for the service endpoint you exposed in the protobuf using your favorite language.

We will use python for this session. There are various tutorials available in numerous language at our dev-portal we have mentioned earlier.

This basically means we have to attach this particular protobuf to the service that we have developed.

The implementation details vary from language to language. But mostly they are consistently based on three things

Compile the protobuf

- This platform agnostic information need to be compiled to a format that needs to consumable by the language of your choice.

The compilation step can be described as:

```
python3.6 -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=.
service_spec/LanguageDetection.proto
```

- Request/Response parameters to attach to the input/response of the application.

In our application the input to the system is rather simple. We give it a sentence. So we would have to wrap the input to the system in form that is similar to that.

Code Snippet

```
from service.language_detection import LanguageDetector
from service_spec.LanguageDetection_pb2 import Input, Output,
Prediction, Language
from service_spec.LanguageDetection_pb2_grpc import
LanguageDetectServicer, add_LanguageDetectServicer_to_server
import grpc
from concurrent import futures
import time


class LanguageDetectServicer(LanguageDetectServicer):
    def infer(self, request, context):
        if request.input is None:
            context.set_code(grpc.StatusCode.INVALID_ARGUMENT)
            context.set_details("Sentence is required.")
            return Output()
        elif request.input == '':
            context.set_code(grpc.StatusCode.INVALID_ARGUMENT)
            context.set_details("Sentence is empty.")
            return Output()

        response = Output()

        detector = LanguageDetector()

        result = detector.language_id(request.input)

        for key, value in result.items():
            lang = response.language.add()
            lang.sentence = key
            for k, v in value.items():
                pred = lang.prediction.add()
                pred.language = k
                pred.confidence = v

        return response


def create_server(port="8001"):
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))

    add_LanguageDetectServicer_to_server(LanguageDetectServicer(),
server)
    server.add_insecure_port('[::]:' + str(port))
    return server


if __name__ == '__main__':
```

```
    server = create_server()
    server.start()
    _ONE_DAY = 60 * 60 * 24
    try:
        while True:
            time.sleep(_ONE_DAY)
    except KeyboardInterrupt:
        server.stop(0)
```

We can test this code by running the test code available with the above code

```
docker exec -it lang /bin/bash
python3.6 -m unittest test_rpc_call.py
```

This would return the following

```
language {
  sentence: "Ich bin das Singularit\303\244t."
  prediction {
    language: "German"
    confidence: 96.0
  }
  prediction {
    language: "un"
  }
}
language {
  sentence: "I am the singularity."
  prediction {
    language: "English"
    confidence: 95.0
  }
  prediction {
    language: "un"
  }
}

Detector is not able to detect the language reliably.
language {
  sentence: "Ich bin das Singularit\303\244t."
  prediction {
    language: "German"
    confidence: 96.0
  }
  prediction {
    language: "un"
```

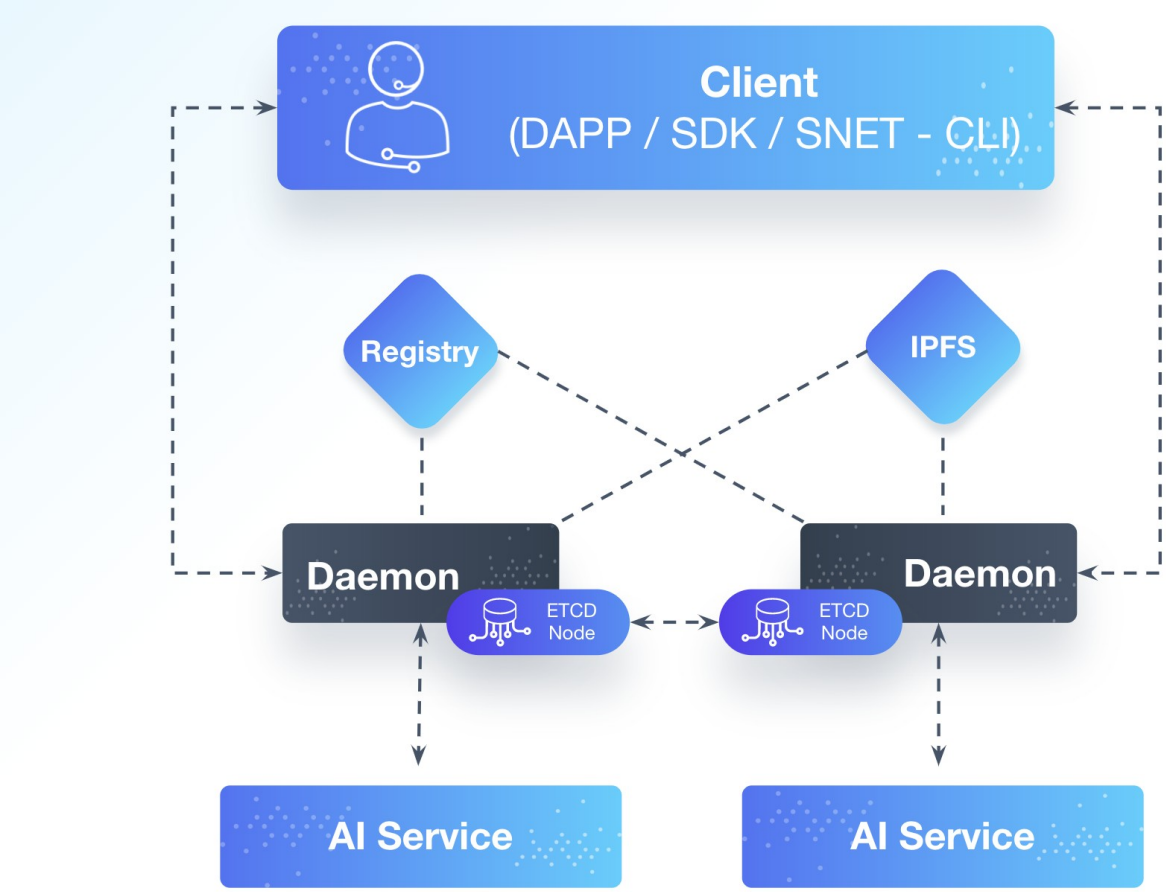
```
  }
}
language {
  sentence: "I am the singularity."
  prediction {
    language: "English"
    confidence: 95.0
  }
  prediction {
    language: "un"
  }
}
language {
  sentence: "\341\212\245\341\212\224 \
341\213\250\341\212\220\341\214\240\341\210\213\341\212\220\341\211\2
65 \341\213\260\341\210\250\341\214\203 \341\212\220\341\212\235::"
  prediction {
    language: "Amharic"
    confidence: 97.0
  }
  prediction {
    language: "un"
  }
}
```

Integrating with SNET-DAEMON

As we just saw one can call any grpc endpoint as long as we have the address. It doesn't have payment logic, authentication or anything that would hamper anyone for using this particular service.

This is where the daemon comes into place.

Lets see the image again:



As you can see based on our architecture the only thing that we need to make sure is the daemon end points to our service.

How do we start with this? We will defer the tools for integration for the next section. What we will now focus on is the code organization that we need to take care off.

Code Snippet

```
import pathlib
import subprocess
import signal
import time
import sys
import argparse


def main():
    parser = argparse.ArgumentParser(prog="run-snet-service")
    parser.add_argument("--daemon-config-path-mainnet", help="Path to
daemon configuration file for mainnet", required=True)
```

```
    parser.add_argument("--daemon-config-path-ropsten", help="Path to
daemon configuration file for ropsten",
                        required=True)
    args = parser.parse_args(sys.argv[1:])
    daemons = {'mainnet':args.daemon_config_path_mainnet,
'ropsten':args.daemon_config_path_ropsten}
    snetd_p = []

    def handle_signal(signum, frame):
        for i,_ in enumerate(daemons.keys()):
            snetd_p[i].send_signal(signum)
        service_p.send_signal(signum)
        for i,_ in enumerate(daemons.keys()):
            snetd_p[i].wait()
        service_p.wait()
        exit(0)

    signal.signal(signal.SIGTERM, handle_signal)
    signal.signal(signal.SIGINT, handle_signal)

    root_path = pathlib.Path(__file__).absolute().parent.parent
    for daemon in daemons.keys():
        snetd_p.append(start_snetd(root_path, daemons[daemon]))
    service_p = start_service(root_path)

    while True:
        for i, daemon in enumerate(daemons.keys()):
            if snetd_p[i].poll() is not None:
                snetd_p[i] = start_snetd(root_path, daemons[daemon])
        if service_p.poll() is not None:
            service_p = start_service(root_path)
        time.sleep(5)


def start_snetd(cwd, daemon_config_path=None):
    cmd = ["./snetd-linux-amd64"]
    if daemon_config_path is not None:
        cmd.extend(["--config", daemon_config_path])
    return subprocess.Popen(cmd)


def start_service(cwd):
    return subprocess.Popen(["python3.6", "start_service.py"])


if __name__ == "__main__":
    main()
```

This is a sidecar pattern:

Note: You can learn more about this pattern and its relative merits https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar

There is installation script the downloads the version of daemon. This is pattern we use for numerous applications.

We will discuss the configuration script that will be used in the next section.

Starting Integration with the Platform

Ok. This is one of last bits of the puzzle.

More information can be found at: https://dev.singularitynet.io/tutorials/publish/

For this part we need to install few dependencies. They need not be part of the image container or be the machine that is where your service resides.

Install snet-cli

```
python3.6 -m pip install snet-cli
```

Check the version

```
snet version
# version: 0.2.11
```

Create Identity

```
snet identity create name_account
```

Get ETH and AGI

- Testnet AGI token can be found at: https://faucet.singularitynet.io/
- Testnet Ethereum coin can be found at: https://faucet.metamask.io/

Go to these sites and get some tokens from your metamask installed container

Choosing network

For the various networks that our contracts are published we have different networks. As of the writting of this we have contracts on the following networks

- Mainnet
- Kovan
- Ropsten

The last two are testnet.

```
snet account balance
```

You should see results similar to

```
    account: 0xd708A068F606893120eA34eECe8a2f7448Fca764
    ETH: 1.987299947853141824
    AGI: 48
    MPE: 9.99889888
```

Creating an Organization

Every user needs an organization/username where their published services would reside in. To create such we would follow the following steps:

```
snet organization create "$INSERT_NAME" --org-id $INSERT_ID -y
```

Something like the following should be printed:

```
Creating transaction to create organization name=test-sg id=test-id

```

```
# gas_price = 1.000000 GWei
    transaction:
        chainId: 3
        data:
'0x9e490710746573742d6964000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000006
0000000000000000000000000000000000000000000000000000000000000a00000
0000000000000000000000000000000000000000000000000007746573742
d736700000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000'
        from: '0xd708A068F606893120eA34eECe8a2f7448Fca764'
        gas: 169287
        gasPrice: 1000000000
        nonce: 134
        to: '0x5156fDE2Ca71da4398F8c76763c41BC9633875e4'
        value: 0

Proceed? (y/n): y
Submitting transaction...

    event_summaries:
    -   args:
            orgId:
746573742d6964000000000000000000000000000000000000000000000000
        event: OrganizationCreated
    receipt_summary:
        blockHash:
'0x703a7c08d4f6f3b337398d4e656a68ea89c77508290f6525a45ad8821a815663'
        blockNumber: 5822830
        cumulativeGasUsed: 4492960
        gasUsed: 169287
        transactionHash:
'0x76ebea160a1b32c901e6448118a1cb2c30649e412ab3d2abe1a8a4e0d21d8ba3'

id:
test-id
```

Note: Standards we use for the organizations names can be found at https://dev.singularitynet.io/docs/concepts/naming-standards/

You should check whether the organization you had created exists by typing:

```
snet organization list
```

The next thing is the most important concept today, it deserves its own section.

Metadata Creation

Metadata is what the daemon uses to store information about the particular service it is currently service as driver for.

Let me show you a sample configuration

Code Snippet

```
{
    "version": 1,
    "display_name": "language-detection",
    "encoding": "proto",
    "service_type": "grpc",
    "payment_expiration_threshold": 40320,
    "model_ipfs_hash":
"QmWrS9XKQcYboLznNkxbVPwuHmwjcTiaYM6qRS84oV7QBB",
    "mpe_address": "0x7E6366Fbe3bdfCE3C906667911FC5237Cc96BD08",
    "pricing": {
        "price_model": "fixed_price",
        "price_in_cogs": 1
    },
    "groups": [
        {
            "group_name": "default_group",
            "group_id":
"NDZ7/arjFKE8JXoh5kQnaX82Zz1qAo/Z+3ylVX/qp4E=",
            "payment_address":
"0xd708A068F606893120eA34eECe8a2f7448Fca764"
        }
    ],
    "endpoints": [
        {
            "group_name": "default_group",
            "endpoint": "https://tz-services-1.snet.sh:8010"
        }
    ],
    "service_description": {
        "description": "Detect the languages in a given string using
polyglot library. Sentences are separated by the tokenization toolkit
of polyglot.",
        "url":
"https://singnet.github.io/nlp-services-misc/users_guide/language-
detection-service.html"
    }
```

```
}
```

We will discuss now what most this parameters are. And how you can create simple configuration that you can modify to your particular choice.

Note: You can go the following link to get more information on this.
https://dev.singularitynet.io/tutorials/publish/#step-7-prepare-service-metadata-to-publish-the-service

The step to create this configuration is again straight forward

```
snet account print
```

Get your current account, you will also find this from your metamask plugin in your browser. Use that to replace the content of “value-from-above”.

Warning: You have to be in the folder where the protobuf we have defined earlier exists.

```
snet service metadata-init service_spec/ "language-detection" “value-from-above” --endpoints http://$SERVICE_IP:$SERVICE_PORT --fixed-price 0.00000001
```

You would get much more information at:
https://dev.singularitynet.io/docs/concepts/service-metadata/

Publish it

Once we have the metadata, we can publish it to the platform in a simple manner using the following command

```
snet service publish --metadata-file service.json
```

service.json must exists in the above. You can save it using the metadata-init mentioned earlier.

We let the registry now. Now lets the daemon know that information to actually interact with it.

Daemon Configuration

```
{
   "DAEMON_END_POINT": "$DAEMON_HOST:$DAEMON_PORT",
   "ETHEREUM_JSON_RPC_ENDPOINT": "https://ropsten.infura.io",
   "IPFS_END_POINT": "http://ipfs.singularitynet.io:80",
   "REGISTRY_ADDRESS_KEY":
"0x5156fde2ca71da4398f8c76763c41bc9633875e4",
   "PASSTHROUGH_ENABLED": true,
   "PASSTHROUGH_ENDPOINT": "http://localhost:7003",
   "ORGANIZATION_ID": "$ORGANIZATION_ID",
   "SERVICE_ID": "$SERVICE_ID",
   "PAYMENT_CHANNEL_STORAGE_SERVER": {
       "DATA_DIR": "/opt/singnet/etcd/"
   },
   "LOG": {
       "LEVEL": "debug",
       "OUTPUT": {
          "TYPE": "stdout"
       }
   }
}
```

Warning: There is a different between DAEMON_HOST:PORT and the grpc values that we have talked about earlier. The GRPC endpoints in our case are PASSTHROUGH_ENDPOINT and port.

Here is my ropsten configuration for my langauge detection:

Code Snippet

```
{
   "DAEMON_END_POINT": "0.0.0.0:8010",
   "ETHEREUM_JSON_RPC_ENDPOINT": "https://ropsten.infura.io",
   "IPFS_END_POINT": "http://ipfs.singularitynet.io:80",
   "REGISTRY_ADDRESS_KEY":
```

```
"0x5156fde2ca71da4398f8c76763c41bc9633875e4",
   "PASSTHROUGH_ENABLED": true,
   "PASSTHROUGH_ENDPOINT": "http://localhost:8001",
   "ORGANIZATION_ID": "snet",
   "SERVICE_ID": "language-detection",
   "ssl_cert":
"/etc/letsencrypt/live/tz-services-1.snet.sh/fullchain.pem",
   "ssl_key":
"/etc/letsencrypt/live/tz-services-1.snet.sh/privkey.pem",
   "PAYMENT_CHANNEL_STORAGE_SERVER": {
          "id": "storage-ropsten",
          "host": "127.0.0.1",
          "client_port": 2381,
          "peer_port": 2382,
          "token": "unique-token",
          "cluster": "storage-ropsten=http://127.0.0.1:2382",
          "data_dir": "etcd/storage-data-dir-ropsten.etcd",
          "enabled": true
   },
  "payment_channel_storage_client": {
          "connection_timeout": "5s",
          "request_timeout": "3s",
          "endpoints": ["http://127.0.0.1:2381"]
  },
   "LOG": {
       "LEVEL": "debug",
       "OUTPUT": {
              "TYPE": "stdout"
           }
   }
}
```

How to start the final pipeline

Run the run-snet-service code with this daemon configuration.

Use this code and give it the above configuration:

Code Snippet

```
import pathlib
import subprocess
import signal
import time
```

```
import sys
import argparse


def main():
    parser = argparse.ArgumentParser(prog="run-snet-service")
    parser.add_argument("--daemon-config-path-mainnet", help="Path to
daemon configuration file for mainnet", required=True)
    parser.add_argument("--daemon-config-path-ropsten", help="Path to
daemon configuration file for ropsten",
                        required=True)
    args = parser.parse_args(sys.argv[1:])
    daemons = {'mainnet':args.daemon_config_path_mainnet,
'ropsten':args.daemon_config_path_ropsten}
    snetd_p = []

    def handle_signal(signum, frame):
        for i,_ in enumerate(daemons.keys()):
            snetd_p[i].send_signal(signum)
        service_p.send_signal(signum)
        for i,_ in enumerate(daemons.keys()):
            snetd_p[i].wait()
        service_p.wait()
        exit(0)

    signal.signal(signal.SIGTERM, handle_signal)
    signal.signal(signal.SIGINT, handle_signal)

    root_path = pathlib.Path(__file__).absolute().parent.parent
    for daemon in daemons.keys():
        snetd_p.append(start_snetd(root_path, daemons[daemon]))
    service_p = start_service(root_path)

    while True:
        for i, daemon in enumerate(daemons.keys()):
            if snetd_p[i].poll() is not None:
                snetd_p[i] = start_snetd(root_path, daemons[daemon])
        if service_p.poll() is not None:
            service_p = start_service(root_path)
        time.sleep(5)


def start_snetd(cwd, daemon_config_path=None):
    cmd = ["./snetd-linux-amd64"]
    if daemon_config_path is not None:
        cmd.extend(["--config", daemon_config_path])
    return subprocess.Popen(cmd)


def start_service(cwd):
    return subprocess.Popen(["python3.6", "start_service.py"])
```

```
if __name__ == "__main__":
    main()
```

Call the service from the cli

https://dev.singularitynet.io/tutorials/publish/#step-10-call-your-service-using-snet-cli

How to get more info about us

Your one stop resource to everything we do can be found at https://singularitynet.io/

Go to our github page: https://github.com/singnet
Go to our community page: https://community.singularitynet.io/
Listen to our podcast: https://singularitynet.io/podcast/
Our Request for AI: https://dev.singularitynet.io/docs/concepts/rfai/
Our XLAB initiative: Shiva Rai will add more info on that. And read about it here: https://blog.singularitynet.io/singularitynet-x-lab-shortlist-436056a2e6d2