QRNG Server Application

Introduction

There is increasingly a higher demand for real random numbers for encryption purposes, simulation and many non-deterministic approximation algorithms.

For the reasons discussed above, a system developed for IT was created with the purpose of distributing random numbers generated by them, and also to distribute to whoever IT wants to. example: UA

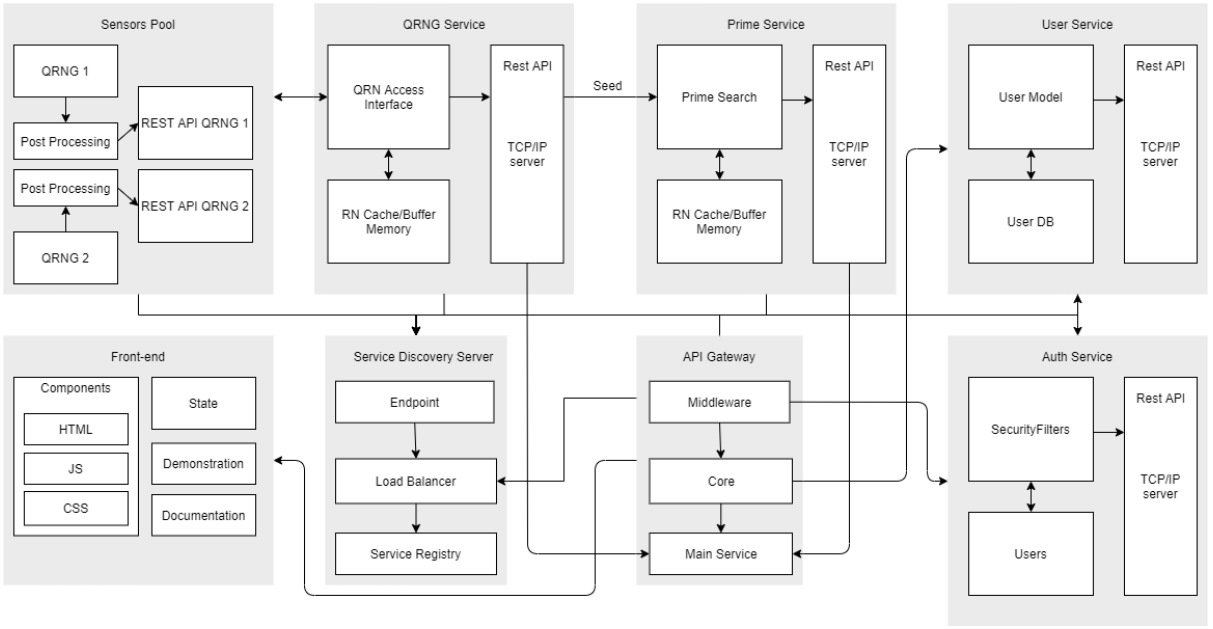
The random values are provided by individual generators however, in order for developers to use these values there should be an abstraction for accessing them. For the previously mentioned reason, a server should distribute the random numbers though a well structured API which serves the requests with configurable levels of security. There is also a need for prime numbers based on these random numbers, possibly for encryption methods.

In order to present the figures of merit of this project a website will also be created which will also provide users with documentation of the API and demonstrations comparing pseudo-random values with real random values.

Requirements

- Easy and fast access to random numbers
- Capability to generate random numbers from different generators at choice
- Provide services through an easy to use API
- Resilience from large spikes of demand
- High throughput
- User management by flexible security
- Services management
- Documentation must be available and easy to understand

System Architecture



The architecture is composed of multiple components:

- **RNG**: registers to the QRNG Service and provides random numbers when requested.
- **QRNG Service**: Abstracts all the Sensors and transforms the random numbers.
- **Prime Service**: Provides prime numbers based on the random numbers from the QRNG Service.
- **Discovery Service**: Provides load balancing capabilities and improves design by removing direct endpoints and translating that to a name which will be converted to an endpoint inside this service.
- **User Service**: Manages the information about the users.
- **API Gateway:** Connects developers to the services though a unique API
- **Auth Service**: Provides authentication based on the User Service
- **Frontend**: Responsible for demonstrating

Services

In this chapter we will discuss in depth how the services were created and why. All services were purposely created to be loosely coupled with each other.

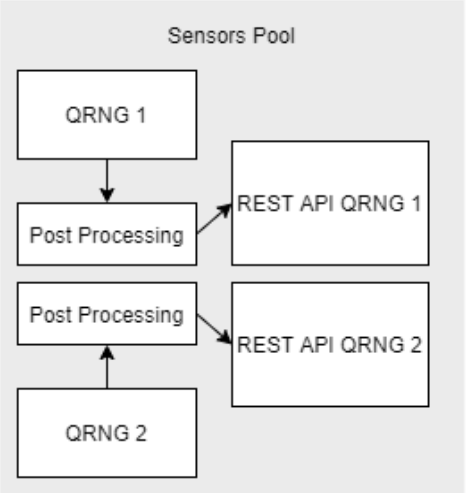
To provide a rapid, frequent and reliable delivery of complex applications, the microservice architecture was chosen. This architecture allows for:

- different teams to work on different services with low communications if any while working together
- loosely coupled code and components
- independently deployable
- highly maintainable and testable

The order in which the services will be reviewed is crescent in terms of abstraction.

All services but RNG were created with java in the Spring Framework, the RNG was created in python because it will be the most used by future users adding generators and python is widely used and easier to use compared to most other languages.

RNG



This service is a wrapper to abstract the generators, for that the service must register in the QRNG Service to be considered a new RNG. The registry is a simple POST request which means that from any platform, no matter the degree of abstraction, it is of trivial implementation. After the registry, the QRNG Service will request random numbers from the RNG at will.

Each RNG implement the following interface which contains 2 methods:

- *registry(machine_name, type_of_RNG)*: POST request to the *QRNG_Service_ip:port/api/generator*. The type_of_RNG can be either *QUANTUM* or *PSEUDO_RANDOM*
- *random(n, bits)*: returns random numbers based on the quantity of random numbers and the length in bytes.

It is recommended that the RNG has a cache to:

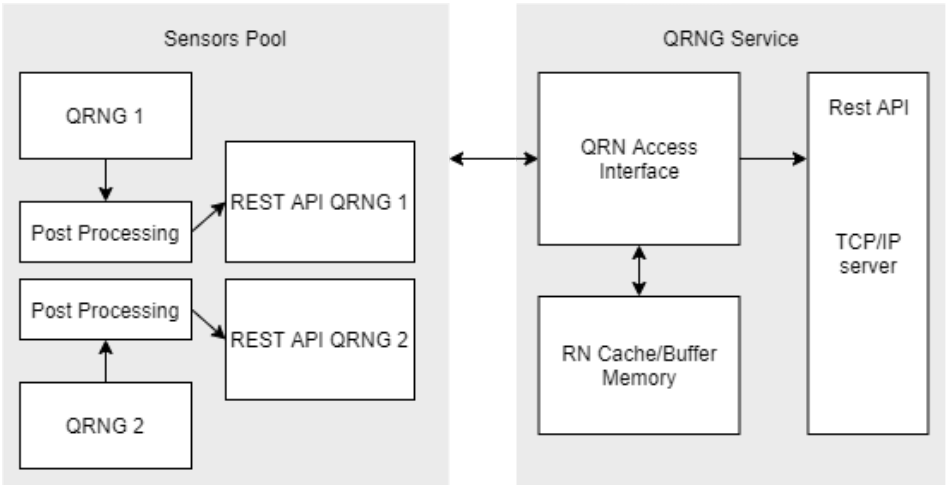
- have smaller response times.
- be more resilient in overwork moments.

The output of the random number is already the final product, when post-processing is needed it is subtended that this service already went through that step.

A different API setup is possible if a new QRNG Service instance which takes this into account is running. This is also possible with the actual Service however, for simplicity sake, only those 2 methods were important enough to be implemented. If ever in need of a quick way of adding methods to this interface the recommendation is to add API version number to each RNG instance and if not present, assume it is the latest.

This service assumes authentication and authorization. It only receives a request and responds with the random numbers it is generating. Therefore it is loosely coupled and easy to change.

QRNG Service



The main purpose of this service is to abstract all the RNGs in the sensors pool. When in need of random numbers this service must be called which internally will request the random numbers from the sensors pool.

For performance reasons, this service has a cache for all the generators it manages. The class that manages this cache is the *RandomNumberInMemoryAccessService* which provides the abstraction to *pop* and *add* bytes from the Sensors Pool stored in cache. The cache is meant to maintain high throughput for short to medium periods of time when the demand is higher than the maximum output rate.

The operation flow depending on the cache is the following: If a request arrives and ...

- there is available cache to fulfill the request:
    1. Asynchronous connection to the RNG(s) is made requesting the same number of bytes it requires from the QRNG Service cache with the *refillRandomNumbers* method from the *RandomNumbersServiceAsync*.
    2. The random numbers are retrieved from the cache and returned encapsulated in a BigInteger object by the *RandomNumberInMemoryAccessService* using the *pop* method because in this repository, the random numbers are stored in a queue.
    3. Collect the random numbers in a list.
- there is some available cache, however not enough:
    1. Asynchronous request to the RNG(s) as previously mentioned.
    2. Synchronous request to the RNG(s) with the remaining random numbers.
    3. Retrieve the random numbers from the cache.
    4. Collect them in a list.
- there is no cache available:
    1. Synchronous request to the RNG(s) with the requested random numbers.

The collection of the random numbers from the generators is paralleled in order to diminish the stacked time of doing the same in a sequential manner.

All these flows still consume a lot of time relatively speaking, for that reason, the handling of generator exceptions is done before, example: confirmation that the generator X exists.

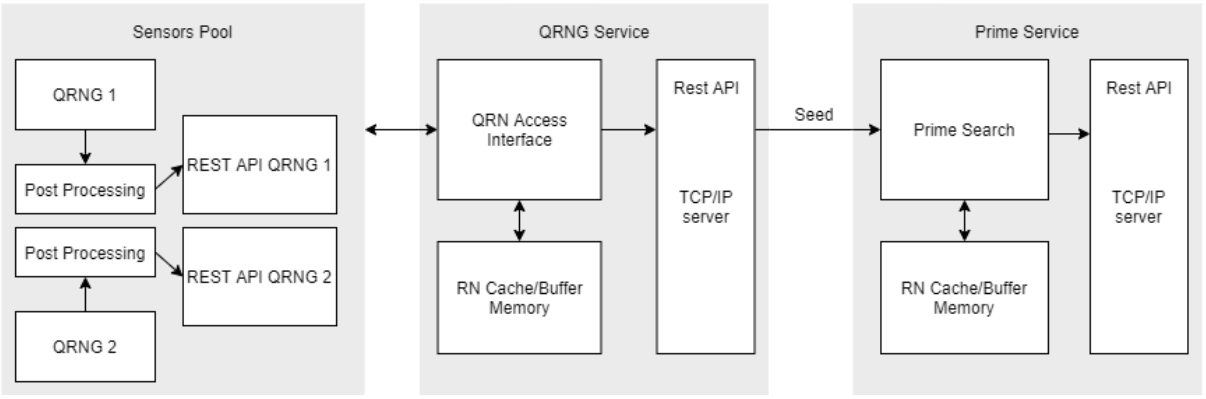
The ability to provide random values constrained by a max and min parameter could have simply coded by discarding the random values which didn't fit in those constraints, however this could be overused by someone simply by clever use of the max and min values in order for the service to continue to work until a random number reaches the desired interval. In order to provide protection against this type of attacks, these same values are passed through a map function which transforms the random values into a different set of random values based on the input parameters, namely the min and max. This assures none of the random bits goes to waste nor accumulates a certain type of biased results.

Parameter	**Function**	**Observations**
max	$n,\ max\ =>\ (n\ \&\ \ldots 0011\ldots)\ \%\ max$	AND operation are useful for performance reasons (remaining calculation are very time consuming)
min	$n,\ min\ =>\ n\ +\ min$	

```
random_values.map( value -> (value + min) % max )
```

For normal to high amounts of requests this cache is enough, if it increases exponentially in the future, the solution is to simply store extra random numbers in disk instead of cache because the main memory is faster, however it’s smaller. A condition would be used to inform the system to read the random bytes from the disk into the main memory when in need and to write the extra numbers to refill the disk. This solution is very unlikely to happen because the round-trip time between the QRNG Service and each RNG will be extremely small due to them being in the same physical facilities, therefore the cap will be determined by the generator, not the memory access.

Prime Service



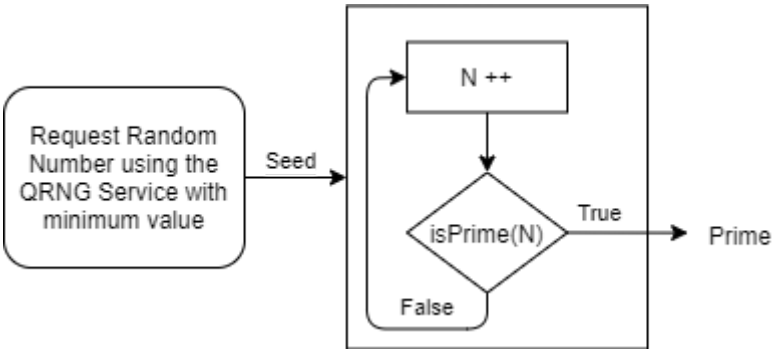
Provide random prime numbers calculated based on random values delivered by the QRNG Service. A lot of algorithms could be implemented to do such a task, the one chosen for the moment is the BigInteger from the Java Math library, the constructor of this class takes the following parameters: bitLength: int, certainty: int, rnd: Random. This returns a probable prime numbers based on the certainty, the lower bound of being prime number is:

$$1 - (1/2)^{certainty}$$

This method is both fast and very reliable with the certainty being above 20, $1 - (1/2)^{20} = 0.99999904632$ which means 99.99994632% certainty is prime.

Another way of computing the next prime number after a random seed obtained from the QRNG Service could be simply incrementing the value and checking if it is prime.

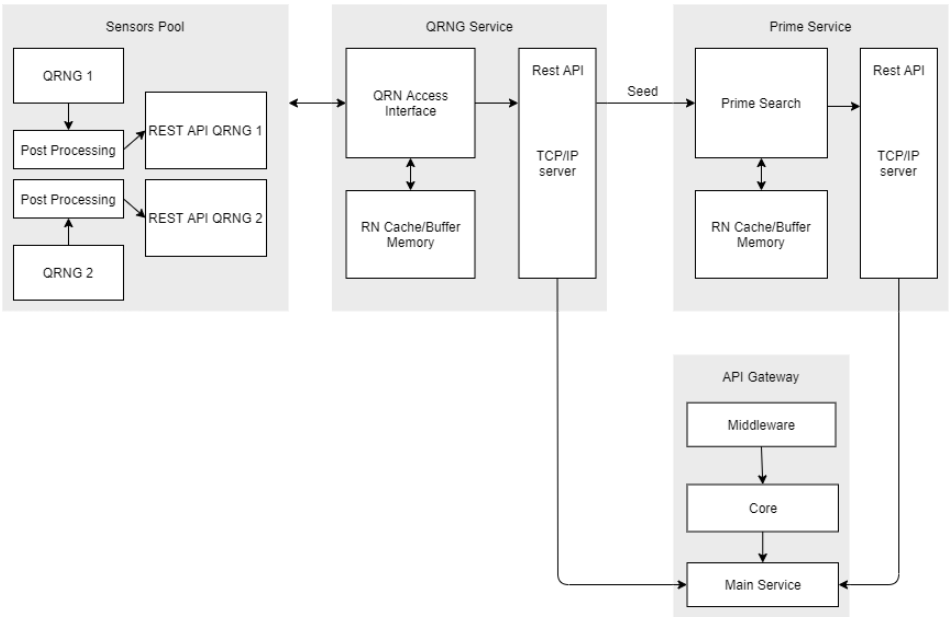
The schematic algorithm could be the following:



The function isPrime performs the *Miller-Rabin primality test* which determines whether a given number is likely to be prime. This function receives an input which provides more probability of being prime the larger the number and with a smaller number, the probability of being prime decreases. This accuracy can be described by at most 4^{-k} and the accuracy of this algorithm is $O(k \times log^3 N)$ where N is presented in the diagram above as the seed and the posterior incrementations.

API Gateway

This architecture already has at least 3 different endpoints, with possible multiple different api structures. The ideal was for the developer to simply use one endpoint and one API, for that reason, an API Gateway was implemented.

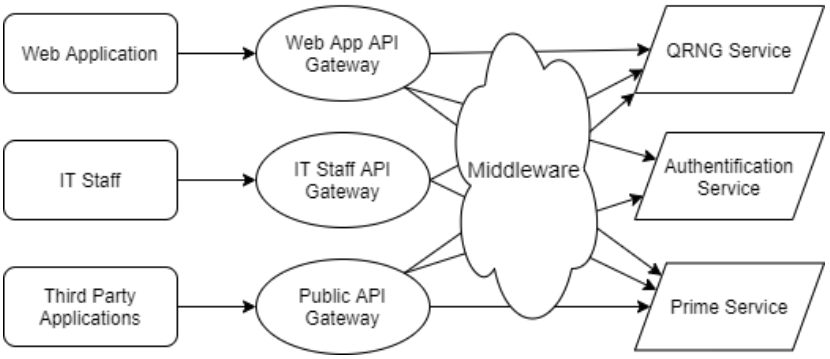


Each service must register to be a part of the cluster, even new instances of the same service are accepted.

A system can have multiple API Gateways, one or multiple for each branch of work and each can have it's own middleware.

All the instances can be inspected to check availability and to perform system monitoring in http://localhost:8761/eureka/.

Another advantage of this service is that the CORS management is done in a single place.



Advantages

- Provide a unique endpoint to developers
- Protect the Service API from abusive users by oculting the real services endpoints
- Maintain compatibility to developers if the service API is updated or switched
- Provide authentication features
- Select API if there is different endpoints for web application, mobile app and third party application
- Provide billing system features for third party applications
- Statistics of usage
- Limiting users for bad behaviour
- Security Policies implementations
- Abstraction of different Services being called in the same request

In order to make good use of the API Gateway, all services will be hidden under this layer for the main Service. All the sub-services will not have API Gateway for performance and simplicity reasons alone.

To have statistics, control and blockage capabilities on users/organizations, all the requests must have an API Key which identifies the requestor. Only known API Keys will be accepted

Requests

There are 4 different requests for the public API:

1. GET: {{ base_url }}/api/random/{{ params }}

Key	Default value	Type	Description
trusted_generators	[]	List	accepted generators
bits	32	integer	number of bits requested
n	1	integer	number of random numbers to be generated
min	1	integer	min value
max	null	integer	max value

2. GET: {{ base_url }}/api/prime/{{ params }}

Key	Default value	Type	Description
trusted_generators	[]	List	accepted generators

bits	32	integer	number of bits requested
n	1	integer	number of random numbers to be generated
certainty	20	integer	certainty of being number prime

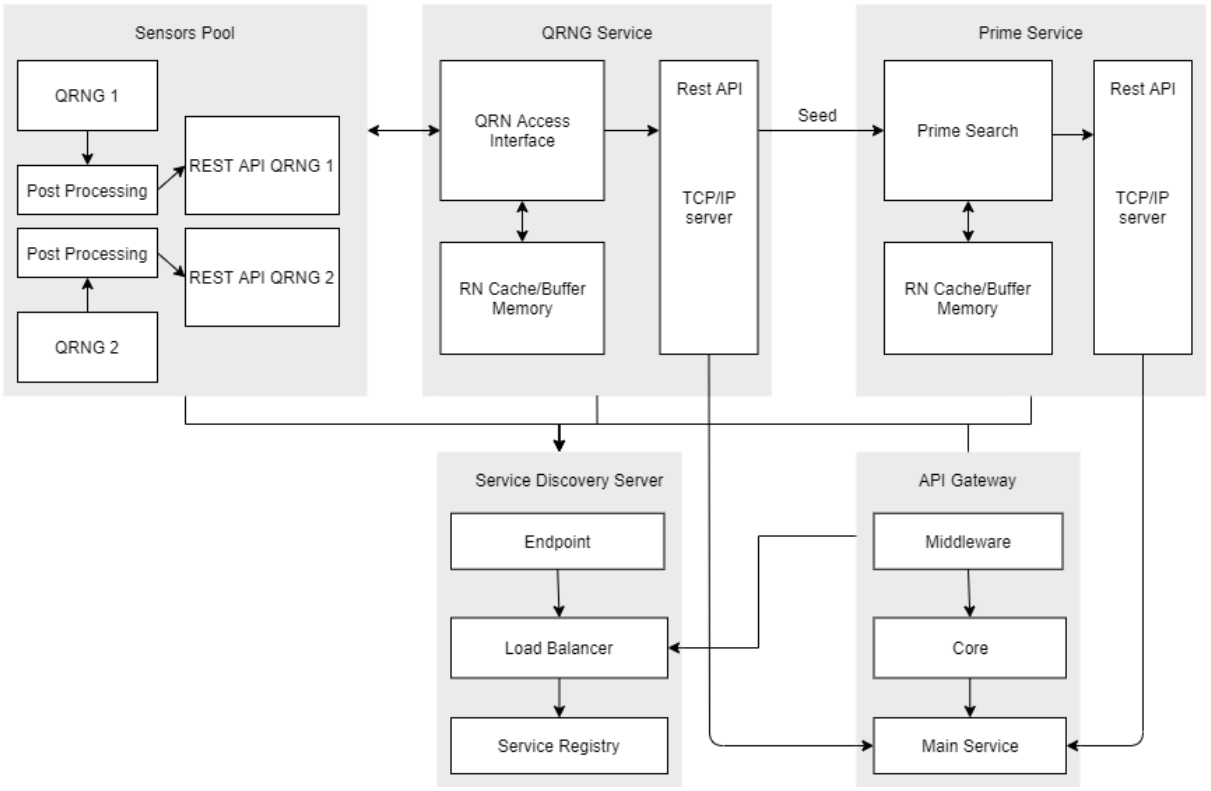
3. GET: {{ base_url }}/api/generator/
4. POST: {{ base_url }}/api/generator/ {{ body }}

name	“QRNG”	integer	number of bits requested
type	QUANTUM	integer	number of random numbers to be generated
url	localhost:8001	integer	certainty of being number prime

Discovery Service

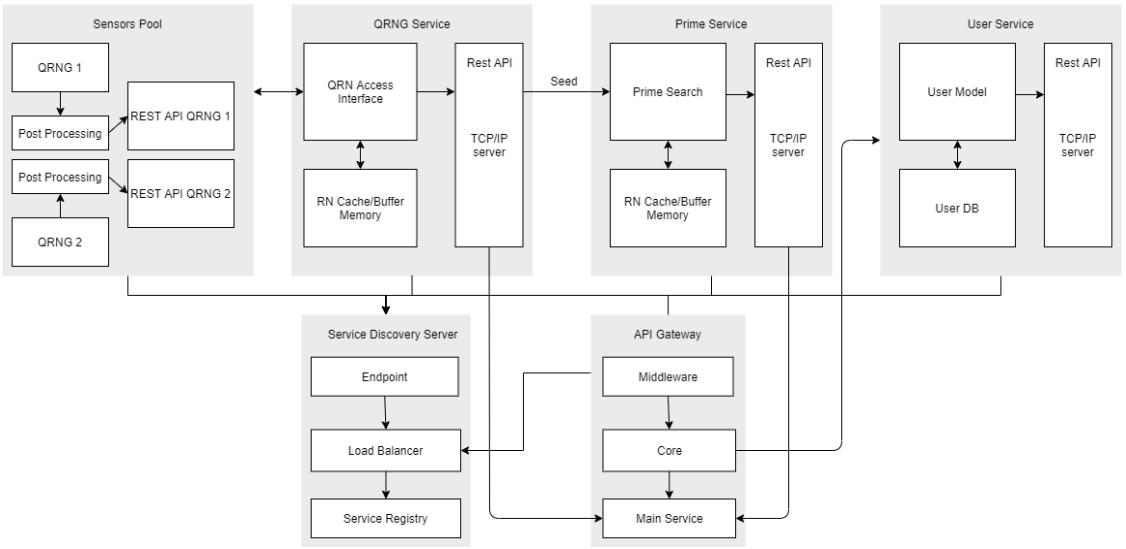
It is never a good option to hard-code the endpoints in which to retrieve or send information because an IP could change, but even if there was a DNS entry for each endpoint, it is still a bad practice, especially when dealing with multiple services, from multiple instances.

For the mentioned above, the discovery service is essential because it provides an easy way of accessing services through service name instead of machine IP. This facilitates a lot of the work of load balancing between machines. Load balancing between instances is valuable because even if one particular part of the system is slower, the ability to change that is by simply instantiating a new instance of that slow service in another machine, that computer will join the cluster and the load balancer will kick off and distribute the load.



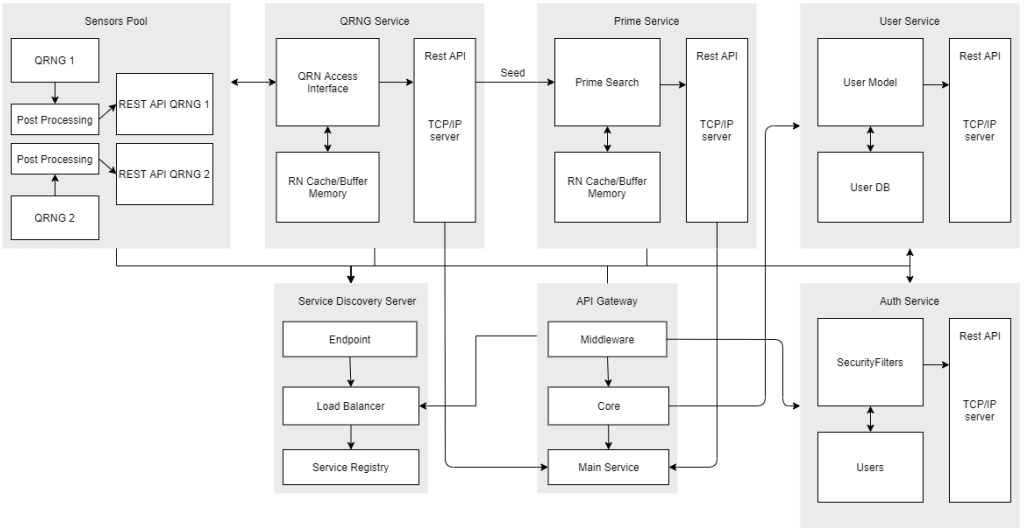
User Service

The User Service focuses on storing the user information without any other worth mentioning aspect for the overall architecture.



Auth Service

This service was the only one that didn't get implemented as a separate service because it is actually running inside the API Gateway Middleware Filters. It retains the auth information and based on that and the information retrieved from the User Service, it evaluates if the user can do or can not do what is trying and prompts a respective error response.



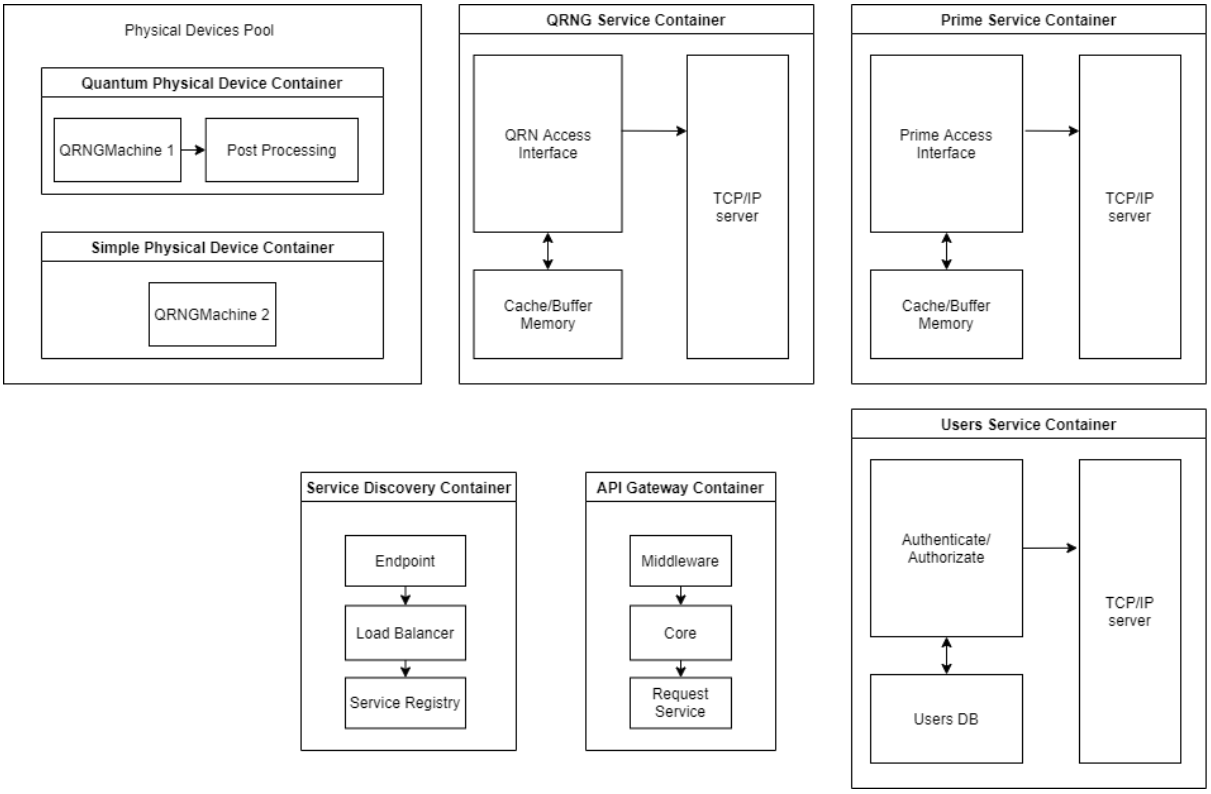
Containerization

Every component of the project will be running on its own container in order to provide cross-compatibility throughout operation systems and to increase deployment speed to every service. The connection between each container is not included in the diagram because the takeaway is that each one is isolated from the rest.

The containerization is provided by docker, a docker-compose file was also created in order to facilitate non-technical personnel to use this system. What it does is to abstract the docker and focus on orchestrating the different containers. If in the future, the computers used to work with this project are overfilled with containers by mismanagement of the containerization, there are 2 ways of dealing with that, removing the containers by the CLI or by the GUI developed by Portainer, this is a free docker container which enables centralized configuration, management and security of Docker environments and Kubernetes, design to allow delivery of containers as a service.

When running all containers in the same machine, a network bridge is needed to convert endpoints for the Service Discovery Service, however when in production, assuming this is a distributed system, this network is no longer needed because the IP and port are enough so there is no need to simulate a network.

On the initialization of each container if the ip of the communications are not explicit, then the localhost and a default port will be used.



How to use

Run each service individually

As said in the previous chapter, containerization was implemented, for that reason there is a need to build those containers

```
docker-compose build
```

If the containers are already created, then it's only needed to run

```
docker-compose up
```

If it is desired to run and build if needed we can run

```
docker-compose up --build
```

if there is a need to put those containers down it is simply

```
docker-compose down
```

It is not recommended to constantly build new images when they are already created because this will fill the disk of the computer and also take time building the image again, so when developing, programmers should get inside the docker container

```
docker exec -it containerID bash
```

and run inside what they want. For that a volume must also be created to connect the code that is in the computer and the code that runs inside the container. When this system is implemented we get the best of both worlds, the OS abstraction of the container and the reliability of the computer in which it is running.

Run all services at once

Moving to the bash scripts, this can be used to start all services at once, the scripts are in the deploy folder and can be built by

```
bash build_containers.bash
```

can be launched by

```
bash launch_containers.bash
```

when there is a need to destroy all the containers

```
bash destroy_containers.bash
```