Design Assessment – PAYPAY

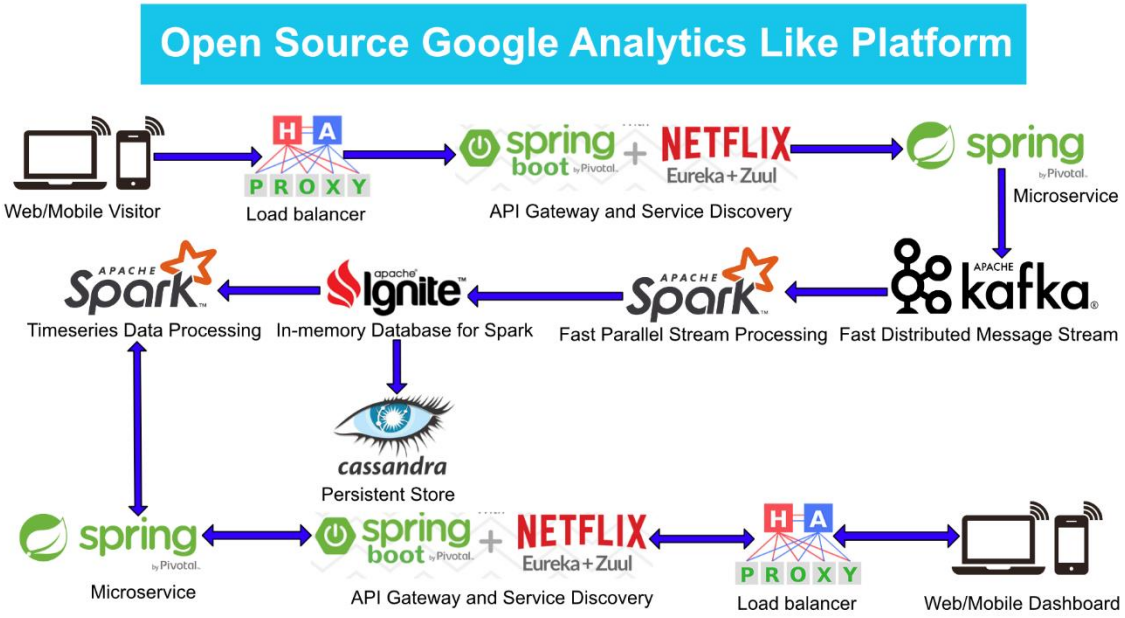
Submitted by Abhilash Krishnan

Table of Contents

Google Analytics (GA) like Backend System Architecture

There are numerous way of designing a backend. We will take Microservices route because the web scalability is required for Google Analytics (GA) like backend. Micro services enable us to elastically scale horizontally in response to incoming network traffic into the system. And a distributed stream processing pipeline scales in proportion to the load.

Here is the High Level architecture of the Google Analytics (GA) like Backend System.



Components Breakdown

Web/Mobile Visitor Tracking Code

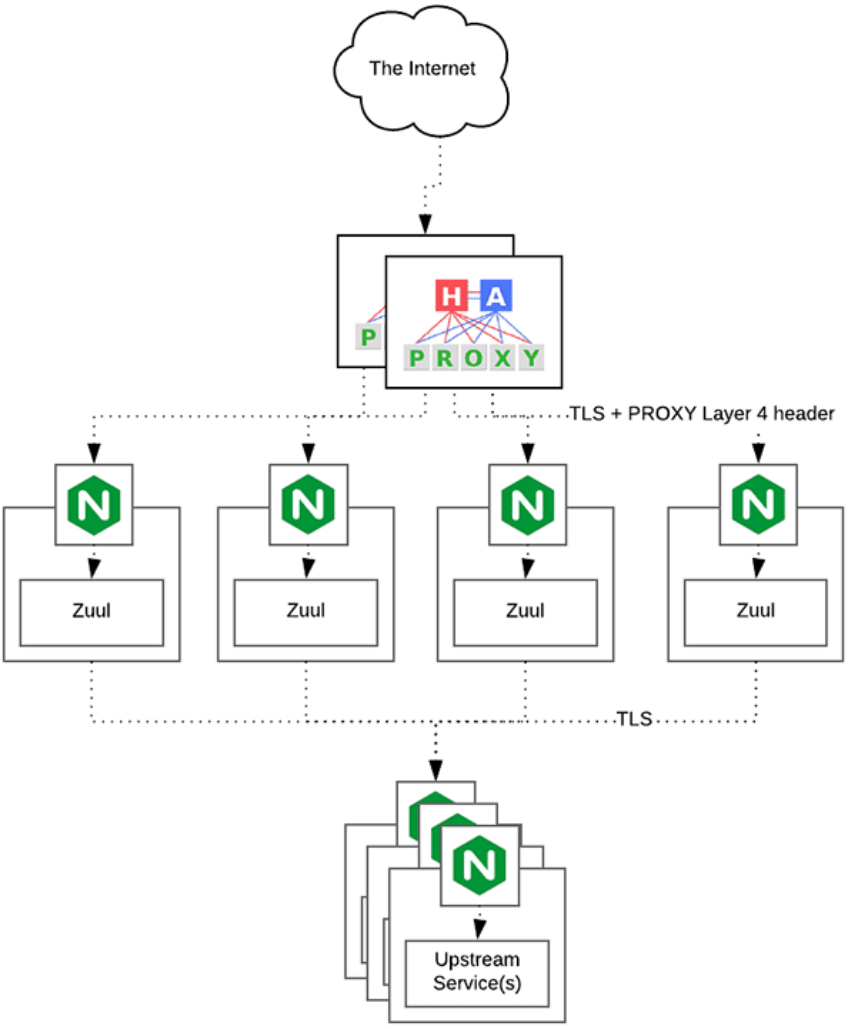
Every web page or mobile site tracked by GA embed tracking code that collects data about the visitor. It loads an async script that assigns a tracking cookie to the user if it is not set. It also sends an XHR request for every user interaction.

HAProxy Load balancer

HAProxy, which stands for High Availability Proxy, is a popular open source software TCP/HTTP Load Balancer and proxying solution. Its most common use is to improve the performance and reliability of a server environment by distributing the workload across multiple servers. It is used in many high-profile environments, including: GitHub, Imgur, Instagram, and Twitter.

A backend can contain one or many servers in it -- generally speaking, adding more servers to your backend will increase your potential load capacity by spreading the load over multiple servers. Increased reliability is also achieved through this manner, in case some of your backend servers become unavailable.

HAProxy routes the requests coming from Web/Mobile Visitor site to the Zuul API Gateway of the solution. Given the nature of a distributed system built for scalability and stateless request and response handling we can distribute the Zuul API gateways spread across geographies. HAProxy performs load balancing (layer 4 + proxy) across our Zuul nodes. High-Availability (HA) is provided via Keepalived.



Spring Boot & Netflix OSS Eureka + Zuul

Zuul is an API gateway and edge service that proxies requests to multiple backing services. It provides a unified “front door” to the application ecosystem, which allows any browser, mobile app or other user interface to consume services from multiple hosts. Zuul is integrated with other Netflix stack components like Hystrix for fault tolerance and Eureka for service discovery or use it to

manage routing rules, filters and load balancing across your system. Most importantly all of those components are well adapted by Spring framework through Spring Boot/Cloud approach.

An API gateway is a layer 7 (HTTP) router that acts as a reverse proxy for upstream services that reside inside your platform. API gateways are typically configured to route traffic based on URI paths and have become especially popular in the microservices world because exposing potentially hundreds of services to the Internet is both a security nightmare and operationally difficult. With an API gateway, one simply exposes and scales a single collection of services (the API gateway) and updates the API gateway's configuration whenever a new upstream should be exposed externally. In our case Zuul is able to auto discover services registered in Eureka server.

Eureka server acts as a registry and allows all clients to register themselves and used for Service Discovery to be able to find IP address and port of other services if they want to talk to. Eureka server is a client as well. This property is used to setup Eureka in highly available way. We can have Eureka deployed in a highly available way if we can have more instances used in the same pattern.

Spring Boot Microservices

Using a microservices approach to application development can improve resilience and expedite the time to market, but breaking apps into fine-grained services offers complications. With fine-grained services and lightweight protocols, microservices offers increased modularity, making applications easier to develop, test, deploy, and, more importantly, change and maintain. With microservices, the code is broken into independent services that run as separate processes.

Scalability is the key aspect of microservices. Because each service is a separate component, we can scale up a single function or service without having to scale the entire application. Business-critical services can be deployed on multiple servers for increased availability and performance without impacting the performance of other services. Designing for failure is essential. We should be prepared to handle multiple failure issues, such as system downtime, slow service and unexpected responses. Here, load balancing is important. When a failure arises, the troubled service should still run in a degraded functionality without crashing the entire system. Hystrix Circuit-breaker will come into rescue in such failure scenarios.

The microservices are designed for scalability, resilience, fault-tolerance and high availability and importantly it can be achieved through deploying the services in a Docker Swarm or Kubernetes cluster. Distributed and geographically spread Zuul API gateways route requests from web and mobile visitors to the microservices registered in the load balanced Eureka server.

The core processing logic of the backend system is designed for scalability, high availability, resilience and fault-tolerance using distributed Streaming Processing, the microservices will ingest data to Kafka Streams data pipeline.

Apache Kafka Streams

Apache Kafka is used for building real-time streaming data pipelines that reliably get data between many independent systems or applications.

It allows:

- Publishing and subscribing to streams of records

- Storing streams of records in a fault-tolerant, durable way

It provides a unified, high-throughput, low-latency, horizontally scalable platform that is used in production in thousands of companies.

Kafka Streams being scalable, highly available and fault-tolerant, and providing the streams functionality (transformations / stateful transformations) are what we need – not to mention Kafka being a reliable and mature messaging system.

Kafka is run as a cluster on one or more servers that can span multiple datacenters spread across geographies. Those servers are usually called brokers.

Kafka uses Zookeeper to store metadata about brokers, topics and partitions.

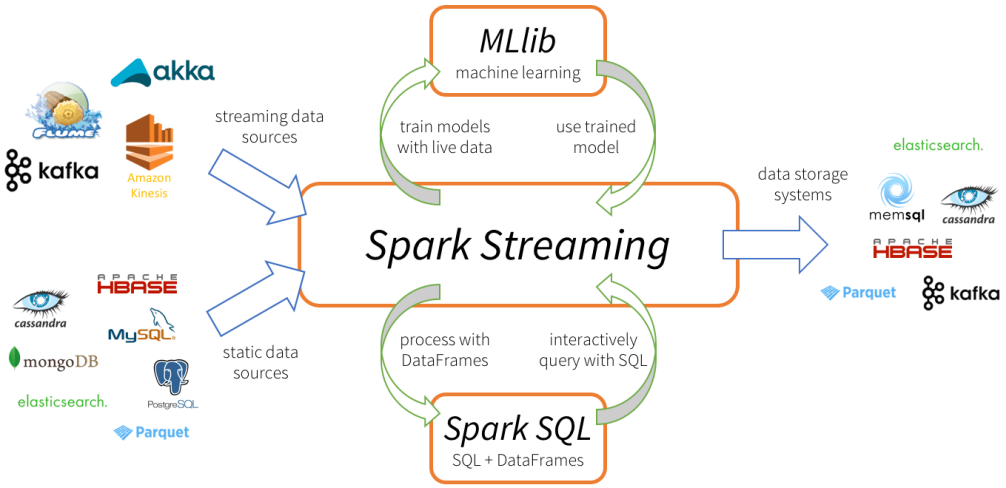
Kafka Streams is a pretty fast, lightweight stream processing solution that works best if all of the data ingestion is coming through Apache Kafka. The ingested data is read directly from Kafka by Apache Spark for stream processing and creates Timeseries Ignite RDD (Resilient Distributed Datasets).

Apache Spark

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

It provides a high-level abstraction called a discretized stream, or DStream, which represents a continuous stream of data.

DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs (Resilient Distributed Datasets).



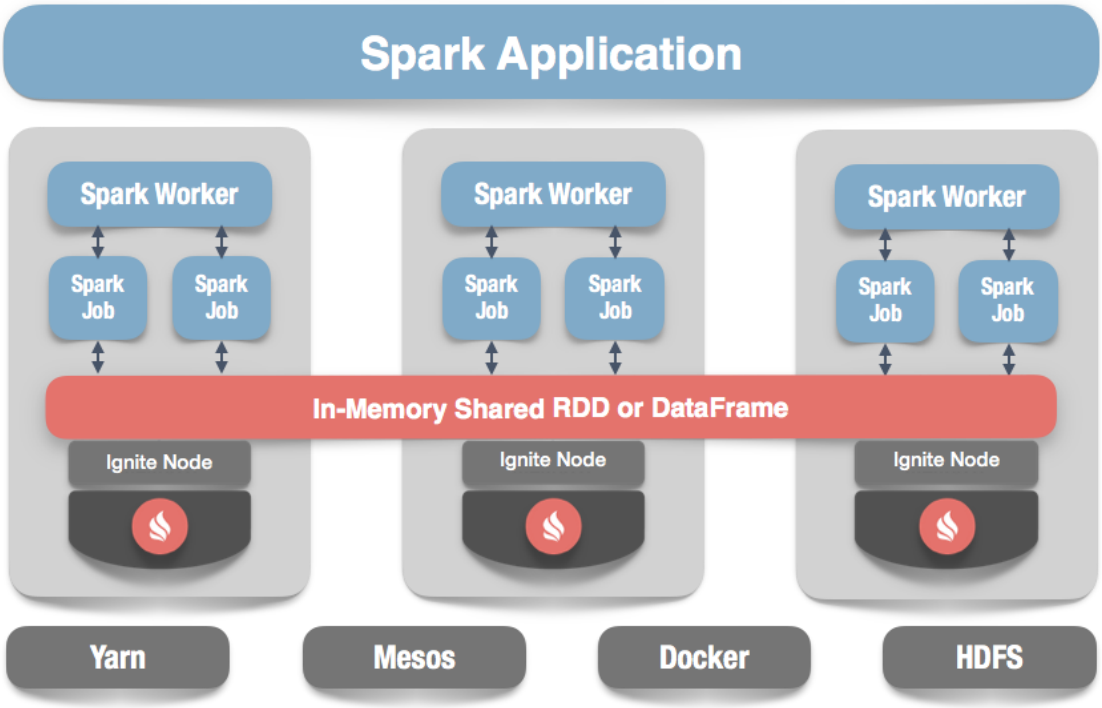
Apache Spark is a perfect choice in our case. This is because Spark achieves high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine.

In our scenario Spark streaming process Kafka data streams; create and share Ignite RDDs across Apache Ignite which is a distributed memory-centric database and caching platform.

Apache Ignite

Apache Ignite is a distributed memory-centric database and caching platform that is used by Apache Spark users to:

- Achieve true in-memory performance at scale and avoid data movement from a data source to Spark workers and applications.
- More easily share state and data among Spark jobs.



Apache Ignite is designed for transactional, analytical, and streaming workloads, delivering in-memory performance at scale. Apache Ignite provides an implementation of the Spark RDD which allows any data and state to be shared in memory as RDDs across Spark jobs. The Ignite RDD provides a shared, mutable view of the same data in-memory in Ignite across different Spark jobs, workers, or applications.

The way an Ignite RDD is implemented is as a view over a distributed Ignite table (aka. cache). It can be deployed with an Ignite node either within the Spark job executing process, on a Spark worker, or in a separate Ignite cluster. It means that depending on the chosen deployment mode the shared state may either exist only during the lifespan of a Spark application (embedded mode), or it may out-survive the Spark application (standalone mode).

With Ignite, Spark users can configure primary and secondary indexes that can bring up to 1000x performance gains.

Apache Cassandra

We will use Apache Cassandra as storage for persistence writes from Ignite.

Apache Cassandra is a highly scalable and available distributed database that facilitates and allows storing and managing high velocity structured data across multiple commodity servers without a single point of failure.

The Apache Cassandra is an extremely powerful open source distributed database system that works extremely well to handle huge volumes of records spread across multiple commodity servers. It can be easily scaled to meet sudden increase in demand, by deploying multi-node Cassandra clusters, meets high availability requirements, and there is no single point of failure.

Apache Cassandra has best write and read performance.

Characteristics of Cassandra:

- It is a column-oriented database
- Highly consistent, fault-tolerant, and scalable
- The data model is based on Google Bigtable
- The distributed design is based on Amazon Dynamo

Right off the top Cassandra does not use B-Trees to store data. Instead it uses Log Structured Merge Trees (LSM-Trees) to store its data. This data structure is very good for high write volumes, turning updates and deletes into new writes.

In our scenario we will configure Ignite to work in write-behind mode: normally, a cache write involves putting data in memory, and writing the same into the persistence source, so there will be 1-to-1 mapping between cache writes and persistence writes. With the write-behind mode, Ignite instead will batch the writes and execute them regularly at the specified frequency. This is aimed at limiting the amount of communication overhead between Ignite and the persistent store, and really makes a lot of sense if the data being written rapidly changes.

Analytics Dashboard

Since we are talking about scalability, high availability, resilience and fault-tolerance, our analytics dashboard backend should be designed in a pretty similar way we have designed the web/mobile visitor backed solution using HAProxy Load Balancer, Zuul API Gateway, Eureka Service Discovery and Spring Boot Microservices.

The requests will be routed from Analytics dashboard through microservices. Apache Spark will do processing and querying of time series data shared in Apache Ignite as Ignite RDDs and the results will be sent across to the dashboard for visualization through microservices.