UNIVERSITAT POLITÈCNICA DE CATALUNYA

BACHELOR THESIS

Big Data Analytics on Container Orchestrated Systems

Author:
Gerard Casas Saez

Supervisor:
Kenneth M. Anderson

A thesis submitted in fulfillment of the requirements for the degree of Bachelor

Developed as part of

Project EPIC Research Group
University of Colorado Boulder

July 5, 2017

Universitat Politècnica de Catalunya

Abstract

Facultat de Informatica de Barcelona

University of Colorado Boulder

Bachelor

Big Data Analytics on Container Orchestrated Systems

by Gerard Casas Saez

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Contents

List of Figures

Listings

1 Introduction

Big Data Analytics is quickly becoming one of the most emergent fields of work. We live in a highly connected society that generates tons of data. Every day, 2.5 Exabytes are moved around the Internet. In addition, it's expected that by 2020, 44 thousand million gigabytes will be the amount of data stored on earth. This constant growth is probably one of the biggest challenges we face in the computer science world. As a consequence, new systems need to be designed to support all of this demand, keeping the response time as well.

On the other hand, containerization is becoming a trend nowadays. Thanks to companies like Docker and its Docker Engine, containers are becoming a de facto standard for software development. It's ease of use and consistent behaviour between systems makes it attractive to developers, as they prefer avoiding difficult and long system setup every time they need to run an application.

With this trend, complexities arrived: difficulties to configure the network and interaction between containers, difficulties to deploy and release new versions and way more. That's when container orchestrated system arrived. They are an abstraction layer over the container management. This systems are in charge of coordinating containers, creating them, destroying them: all the container lifecycle is controlled by the system.

As this systems are becoming more and more popular, software architecture will need to adapt to embrace the new possibilities that this systems provide. Some concepts that previously were close to impossible to do, now are available to any software architect. Applications no longer need to have an static infrastructure, container orchestrated systems can grow and decrease on demand. Systems can be splitted in small microservices, there's no need to merge everything a big monolith application anymore.

In addition, cloud providers are adding support for container orchestrated systems. This makes it even easier to deploy complex infrastructures. Also, you can design systems independently of the providers.

It also helps to find new ways to solve the problem of parallelism and scaling, making it easier to deploy new machines opens the door to new possibilities to have infrastructure on demand or to scale automatically.

2 Background

Before digging into the project itself, we need to understand the problem area.

2.1 Project EPIC

EPIC (Empowering the Public with Information in Crisis) is a project at the University of Colorado Boulder. Its focused in crisis informatics, an area of study that examines how people use different technologies to interact during emergencies and their impact on emergency response.

Project EPIC has a long history on software engineering. Since 2009, this project has done research on large data storage and analysis. When large amounts of data are being collected, software engineers need to focus on structuring this data so it's easy to perform analysis . In addition, this data needs to be easily accessible for analysts.

The research group has with it's own Big Data analytics system running. Developed over the past 8 years by different software engineers researchers, the mentioned system counts with 3 different parts Collect, Analyze and Analytics. EPIC Collect is the in-house Twitter Streaming client that works 24/7 retrieving tweets from the various events that need to be monitored in real time. Since 2009 this software has been collecting tweets with an uptime of 99%. On the storage layer, we have Cassandra. This NoSQL database is focused on writes and allows a really high input. On the other side we have EPIC Analyze, built to help analysts access the data that has been collected on Cassandra. Finally EPIC Analytics is a large machine where analysts execute intensive processes over the collected data.

2.2 Containerization

We understand containerization as operating-system-level virtualization. This technology allows program to be run inside a system, isolated from each other and the host system. It works similarly than traditional virtual machines from the point of view of a program, but makes use of the underlying system resources instead of running a full powered operating system. It provides an abstraction to for programs to run in.

There are many containerization platforms around, however the most used and the one we will use for this project is Docker. This platform was originated as an internal tool in a PaaS company and later on open-sourced. Thanks to many contributions from different companies, this project grew becoming the most used containerization software. It's able to run on almost any system and has many integration. In addition, almost all container orchestrator systems support Docker containers

2.3 Container orchestration technologies

When containerisation technologies raised due to a migration to microservice architectures, big companies needed a way to manage their containers in a more friendly way. Interconnecting containers, managing their deployment, scaling easily, all this tasks were possible with containers, but they were really difficult. This is why container orchestrations systems were born. In order to manage container, the mentioned systems add an abstraction layer on top of the systems, making such tasks easier to perform.

There are a few available container orchestration systems available at the moment. The most popular ones are Kubernetes and Apache Mesos. In this project, we will further explore Kubernetes. The main reason for this is that thanks to its open source community it's easier to find tutorials and courses. In addition, there are a lot of big companies that are backing this project and contributing to it, which provides a security of a long-term support.

Google Cloud seems like the best fit to host Kubernetes as it has a managed cluster option where we don't need to worry to install and wire up the system. In addition, thanks to Google being part of the maintaining team for Kubernetes, there's a great support for Google Cloud infrastructure on Kubernetes.

2.4 Microservices architecture

Microservices is an approach to distributed systems that promote the use of small services with specific responsibilities that collaborate between them, rather than big components with a lot of responsibilities that make interaction more difficult.

In addition, thanks to new technologies like containers and container orchestrated systems, they are quickly becoming a standard on the design of big software systems. This type of architecture has been adopted by many companies as it provides a more maintainable model.

Another of the key advantages of Microservices is that it makes it easier to adopt different technologies for each component. In addition to containerization it decouples differents parts of a system systems. This allows a better optimization of technology depending on the feature that each microservice needs to offer. For example a microservice that needs to store a highly related data can make use of separate graph database, exposing a REST API, allowing for a completly abstraction. At the same time a microservice that needs to store big documents can use a document store database underneath, allowing for a better optimization.

Having small microservices do specific tasks, makes development cycles faster and more independent. Making incremental deployment way easier and less dangerous. They also make it easier to scale systems. We could individually scale parts of the system independently depending on their usage.

Microservices avoid falling in the same problems that previous service-centered architectures had, by centering on their architecture and letting the architect decide the messaging system or deployment type.

In addition microservice architecture allows for a better reliance, as each microservice is independent, if one fails, the rest can remain unaffected. This finds a perfect match with container orchestration systems, as they can abstract health checks on microservice as well deploy again microservices that failed.

In terms of designing microservice architectures there are a couple of approaches that are equally used at the moment: coreography and orchestration.

2.4.1 Coreography

Coreography approach centers all interaction on a publish-subscribe philsophy. Events are propagated through the system in an asyncronous basis. When something happens in a microservice, it gets published into a public queue. Other microservices can subscribe to this queue and act consequently when they receive an event.

This approach makes it the system more flexible, removing the responsability of knowing who to send messages to from the microservices. We can add more components easily without having to modify the existing ones.

However, we need extra components in order to make sure that all tasks are performed once an event is published. We can acomplish this by installing and mantaining a monitoring solution.

2.4.2 Orchestration

Orchestration centers interactions in request/response iteractions. Each microservice requests information to any other microservice. For this approach we need of a proper service discover backend. In addition if the service is down, we need to make sure that each microservice retries and is capable of failing gently.

The good thing is that we would be sure that all actions are performed by the end of an action. However if a components takes too long to answer we could break the entire action.

A more modern approach to orchestration has been developed with Service meshes. This components take the responsibility of balancing requests as well as allowing for service discovery to happen, they avoid bad crashes by retrying requests. They work by adding a sidecar to all microservices acting as proxy between the microservice and the outside network. Together with a manager that controls service discovery, load balancing and retrying counters. However these are still new technologies being developed and are still on their early releases. In addition they don't include the choreography extensibility capabilities.

2.5 Messaging systems

Messaging systems organize queues of messages produced by some microservices and notify subscribed systems that are consuming queues when they arrive. Their main objective is to can decouple components and to have a cache if the consumers can't digest all the incoming messages. This allows for more reliable systems as

system don't depend on the mutual availability to pass messages as they would if they had to use other messaging system like HTTP.

There are many options used nowadays the most popular being Apache Kafka and RabbitMQ. Kafka, is a decentralized, high availability system that allows us to publish messages organized by topics. It allows for high parallelizability thanks to the partitions on topics, which lets you read and write at a higher rate in a parallel way. Each partition can have one consumer associated to it, which makes it faster to read.

Another good thing about Kafka is its persistence system. Thanks to a great integration with the kernel, it performs and persists data faster than other systems. This is due the fact that gives the responsibility to flush to disk to the kernel, taking advantage to disk page caching and memory caching implemented into current operating systems.

In addition, Kafka it's one of the most used messaging systems in the industry nowadays. Many companies use it on their production systems to decouple their systems and increase code responsability repartition. In addition it has a really strong community behind it, as well as some major technology companies like Linkedin maintaining it. Other messaging options include RabbitMQ or NATS.

Messaging systems are needed for choreography microservice architectures, as they are the communication layer between microservices.

2.6 Big Data storage systems

We will be tracking Twitter with the Streaming API. As the API is limited to provide a 10% of the total tweets generated in twitter every minute, we can estimate how many tweets will go through our system. Last report available of tweets per seconds is from 2013. The rate was 5700 tweets[2] per second on average. Which means that our system should be able to store 570 tweets every second. In addition we need a platform that scales well with our increasingly large datasets. As previously studied in the EPIC project[3], we would need to use a NoSQL database instead of a relational database to support the high throughput of tweets and make the system more scalable.

On the other hand we also need a system that allows analytical and operational queries happening at the same time. We would like to be able to analyze our data in real time without having to stop its storage. Given that some analysis may take a few minutes due to the high amount of items to analyze, we need to have a system that allows a high throughput for both parts. In this case Cassandra is the best option as the number of operations per seconds scales the most compared to other NoSQL alternatives, especially with operational and analytical workloads[1].

To store tweets we base our table structure on the current EPIC Analyze structured data storage in Cassandra as we can see on the CQL in the next page. In addition we also add an index on the event_name attribute so that we can access per event faster. We need to add the index as a lot of queries are performed per event, and so we would benefit from accessing tweets from each event in the fastest way.

Listing 2.1 Tweets CQL table

```
CREATE TABLE twitter_analytics.tweet (
    id uuid,
    t_id text,
    event_kw text,
    event_name text,
    hashtags list<text>,
    media_url text,
    t_coordinates text,
    t_created_at timestamp,
    t_favorite_count int,
    t_favorited boolean,
    t_geo text,
    t_is_a_retweet boolean,
    t_lang text,
    t_retweet_count int,
    t_retweeted boolean,
    t_text text,
    u_created_at timestamp,
    u_description text,
    u_favourites_count int,
    u_followers_count int,
    u_friends_count int,
    u_geo_enabled boolean,
    u_id text,
    u_lang text,
    u_listed_count int,
    u_location text,
    u_name text,
    u_screen_name text,
    u_statuses_count int,
    u_time_zone text,
    u_url text,
    u_utc_offset int,
    um_id text,
    um_name text,
    um_screen_name text,
    urls list<text>,
    PRIMARY KEY (id, t_id))
```

2.7 Software development in microservices architectures

Thanks to container-orchestrated systems and the popularization of container systems, software development is easier. With microservices, focus moves out of one component software architecture to organizing the whole system. Now, we can focus on packaging alone components instead of having to packaging all of them at once, which makes the development more flexible. This would allow for a more continuous development in the system allowing for a better performance optimization, and an easier way to keep the system updated with the last set of technologies.

3 Development

3.1 Problem Statement

The goal of this project is to create a system with a better scalability and reliability compared to the existing EPIC infrastructure using a container-orchestrated infrastructure while improving our development cycle by switching to a microservices architecture. In addition to improve real time query flexibility and overall performance.

3.2 Approach

Container orchestrated systems help deploy microservices architectures as well as abstract their physical structure. They add a logical abstraction on top of the physical structure allows to work independently from the underneath infrastructure, which makes it easier to move between different cloud infrastructures.

In addition, microservice architectures make systems more flexible as they decouple between functionalities, making it easier and more cost efficient to replace tools and evolve the system. We will take an asynchronous/choreography approach for microservices with message queuing systems. This approach decreases coupling between components increasing the capability to add new components without having to change any configuration.

We will design the system to fit and take advantage of container orchestrated technology. This systems make it easy to create infrastructure on demand. Which in turn would allow for a better optimization of resources. One improvement of the "on demand" feature, is the ability to modify infrastructure on state change. Which in that sense would make possible to remove the state from the designed microservices, make them unaware of the state changes. Thanks to this capability we can design microservices that are completely stateless, improving the performance by removing responsibilities and making their development and maintenance easier.

As we have previously seen, the current EPIC infrastructure use cases can be summarized in X points:

- Collect tweets in real time
- Classify collected tweets in real time
- Modify classification in real time
- Analyze collected data

In that way, we first need a place to store tweets. To store tweets we use Cassandra as it's the one used by the current EPIC infrastructure. Currently Cassandra is the best

option thanks to it's write perfomance, high scalability and great concurrency. In addition, it's backed by DataStax a private company that has written a high amount of documentation in how to optimize Cassandra deployments.

To collect the data in real time, we need a microservice that connects to the public Twitter API and requests tweets. For that we use the Streaming API. We create a specific microservice for this. In order to remember it later, we will call it Twitter tracker. It will act as a gateway pulling data from the public API and passing it to the messaging system to be processed. We just want to get the tracked tweets into the system.

To classify the incoming data, we can take advantage of the orchestration system and create an instance for each category we need to store. We plug each instance into the queue created by Twitter tracker and make that each instance decides if each message is part of their own category or not. If the answer is positive, this microservice will store the tweet as well as normalize it. This way, we have normalized tweets which in turn will make it easier to analyze the data that we have. We will call this microservice Tweet normalizer. For each category we will have an instance running. As we previously stated, this makes the microservice way easier to develop.

Next, we need to modify and create the classification configuration in real time. To do so, we would need a couple of components:

- *Event Manager UI*: Let's analysts manage categories/events adding keywords, toggling tracking and more. This microservice would not manage anything from the system, it would be completely unaware of the surrounding system where it is running and what other microservices exist. Finally, any change performed in the state will be broadcasted through a messaging system so that any other microservice can act upon the change. Messages should contain full state instead of incremental states. This is to ensure that state storage responsibility is not replicated in differents components.
- *Infrastructure Controller*: Subscribes to the messages generated for events and modifies the current infrastructure creating or destroying instances of tweet normalizer or updating *Twitter tracker* in order to match the change. This microservice needs to be aware of the orchestration system where its running. However, thanks to the *Event Manager UI* sending messages with full state, we can make this component totally stateless. In addition, the responsibility of detecting what changes need to be performed in the infrastructure can stay into the container orchestrated system. So this microservice responsibility stays limited to translating changes in state to changes in infrastructure

Finally, Analysts want to access the collected data and perform analysis on top of it. Cassandra is not really prepared to answer specific queries from analysts. Its CQL interface is great for small queries but doesn't allow for high complexity. In addition, it was never intended to be used as an analysis tool. So we choose a different tool that it's optimized to read and process large datasets: Spark. It is a well known big data processing tool, and it has a connector for Cassandra that takes advantage of the cluster data locality to perform tasks. This way we take the read aspect of the cluster away from Cassandra and give to a tool that has been optimized for this task.In order to make it easier for analysts to access and perform analysis, we will add a web Notebook UI to give analysts an easy way to execute queries on Spark with a web UI.

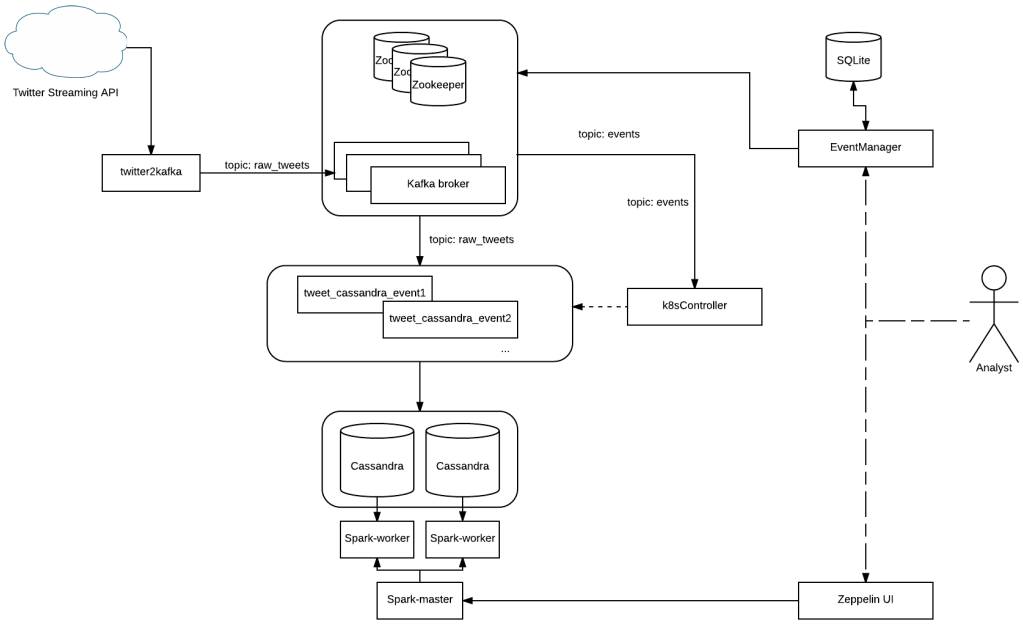

FIGURE 3.1: Proposed system architecture

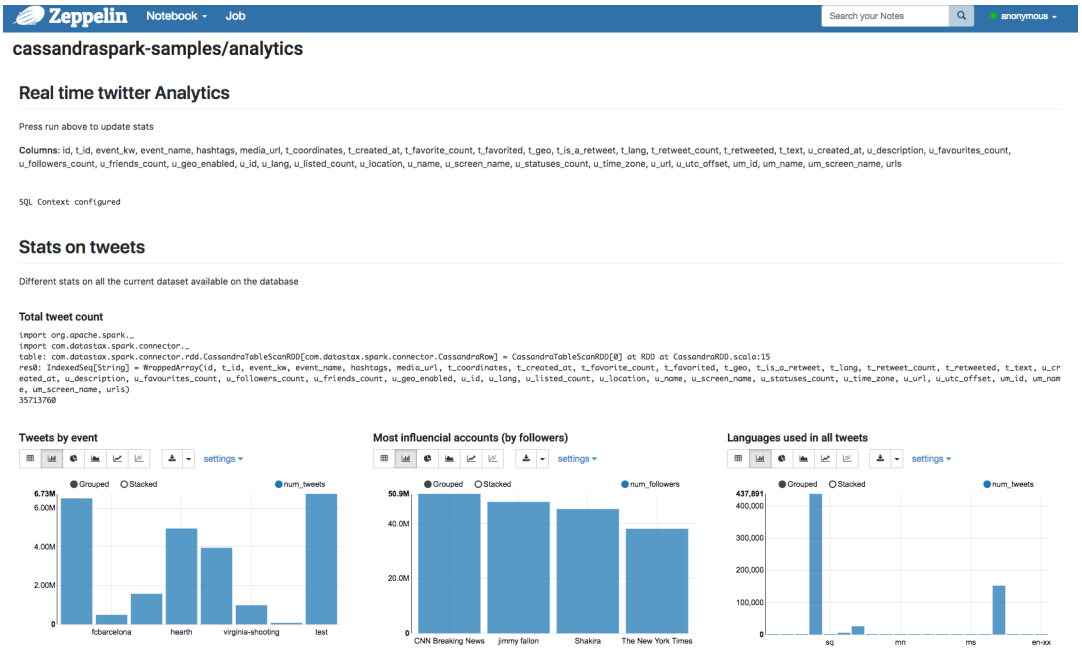

FIGURE 3.2: Zeppelin server with some visualization from the dataset

3.3 Implementation

To prove that the specified architecture is feasible, I developed a working prototype of the system.

We started with 2 nodes of Cassandra to store the tweets with Google Cloud disks attached to keep data persistent. In order to deploy this, we use stateful sets, Kubernetes objects designed to support stateful components and allow scalability. Persistent volumes can be attached to any node to keep data between restarts keeping a naming consistency. They can be used for stateful scalable apps like distributed databases. They provide an easy way to add or removed nodes from the set, allowing for a controlled scalability. In our case we could scale Cassandra dynamically to increase storage space and/or performance by making the cluster more parallel.

In addition, we add a spark workers to each instance of Cassandra to increase and ensure data locality when running Spark jobs. This way, spark works with local data from their correspondent Cassandra node. All of this workers are coordinated by a spark master node that is in charge of planning and distributing the work between each worker.

To interact with the Spark cluster, we deploy a Zeppelin notebook container. We tweaked this container to interact efficiently with our Spark and Cassandra stack by adding a default configuration to use the cassandra-spark library. Notebooks makes the system more accessible by abstracting how to execute scripts in Spark and allowing to present results in an easy format of a website.

We also need some custom microservices for the designed solution. At the beginning all microservices were implemented in Python, basically because it's easy and fast to prototype, and it also let's use iterate faster. The goal was to keep all microservices short and simple in order to make their maintenance easy, enabling developers to

maintain and deploy changes faster. Also, keeping them small makes it easy if any microservice needs to be rewritten in another language because python is not fast enough. Apart from implementing them, we also needed to containerize them to make deployment in the orchestration server easier.

For the collection pipe, we divided the process to support better scalability and avoid high incoming tweets to collapse the system. Between parts we use Kafka a high reliable messaging system. Thanks to this decoupling we could also plug other backend to analyze tweets in real time.

Finally, to make the system work we need to add a few custom components. The following components were implemented specially to work on the described environment.

- *Tweet normalizer*: Fast classifier and tweet data normalizer that receives tweets from the raw_tweets pipe and stores them on Cassandra. This microservice is designed to only keep track of one event. The tracked event information is sent to the microservice on start through environment variables defined with Kubernetes. To increase this microservice performance, some C based libraries replaced some regular Python libraries for intensive task as json encoding and loading.
- *Twitter tracker*: Twitter Streaming API client. Connects to the public Streaming API to fetch tweets and send them to the raw_tweets pipe. Written initially on Python and re-written in Go afterwards to increase throughput and avoid some python errors that appeared after a couple of days running.
- *Event Manager UI*: Stateful django web application for analysts to interact with. Enables CRUD actions on events. Any change on state is later broadcasted to the rest of the system through Kafka. It keeps track of changes made on events to facilitate collaboration between analysts. SQLite as a file is used for storing data and a Google Cloud disk is used to allow saving state between container restarts.
- *Infrastructure controller*: In this case this is a Kubernetes controller, written in python using a python client library. It subscribes to the Kafka topic and receives events generated by the event manager UI. It applies changes to the infrastructure incrementally to meet the changes performed in the UI by the analysts. It's completely stateless, all it does is act upon what is received.

3.3.1 Deployment

To deploy the system, we create a 4 node cluster on Google Container engine. Each node having 1 virtual CPU and 4Gb or RAM. This is the major factor limiting our deployment. We also limit each Cassandra node to 1 GB of RAM for the Cassandra part. We won't limit Spark, as we want it to use as much resources as we have available.

To start we need to deploy the basic components driving the infrastructure. Those are Kafka and Cassandra. In our case we are running Kafka with 2 brokers and Cassandra with 3 nodes each one running Spark as a sidecar. After this we need to deploy the frontend components. One the analytics side we deploy Zeppelin, and on the collecting side we deploy the EventManager. The final part is the Kubernetes

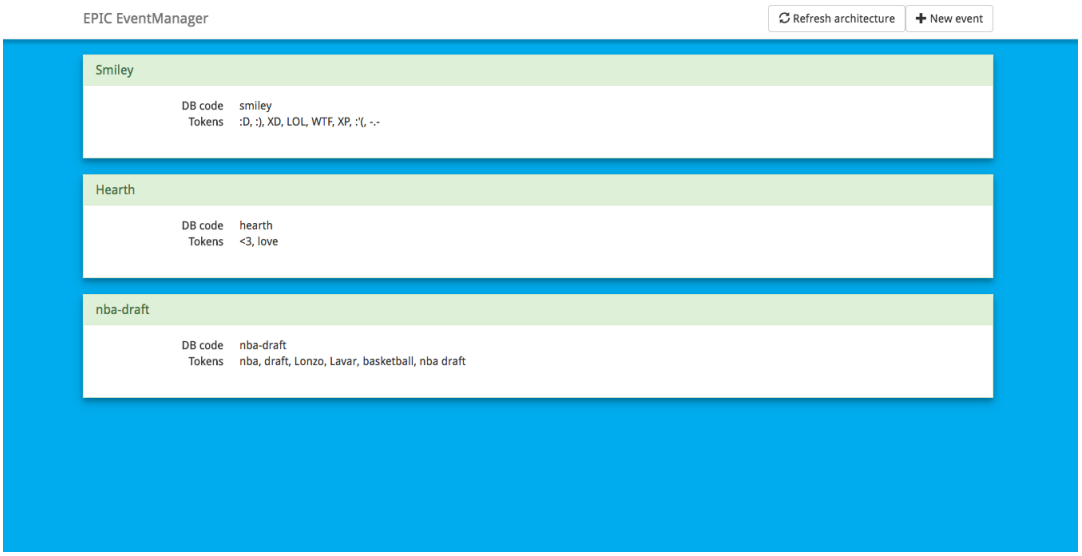

FIGURE 3.3: Event manager UI

controller. Once deployed the rest of the infrastructure will be deployed when the state requires it.

Tracking will start once we add an event in the EventManager UI. After the creation action, the UI will send a message through Kafka saying that this event was created. Kubernetes Controller will collect this message and update the twitter tracker and will create a new microservice to normalize and classify tweets for the provided event.

A Frequently Asked Questions

A.1 How do I change the colors of links?

The color of links can be changed to your liking using:

`\hypersetup{urlcolor=red}`, or

`\hypersetup{citecolor=green}`, or

`\hypersetup{allcolor=blue}`.

If you want to completely hide the links, you can use:

`\hypersetup{allcolors=.}`, or even better:

`\hypersetup{hidelinks}`.

If you want to have obvious links in the PDF but not the printed text, use:

`\hypersetup{colorlinks=false}`.

Bibliography

[1] EndPoint. *Benchmarking Top NoSQL Databases: Apache Cassandra, Couchbase, HBase, and MongoDB*. Tech. rep. EndPoint, 2015.

[2] Raffi Krikorian. *New Tweets per Second Record, and How!* URL: https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html.

[3] Aaron Schram and Kenneth M. Anderson. "MySQL to NoSQL: Data Modeling Challenges in Supporting Scalability". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. SPLASH '12. Tucson, Arizona, USA: ACM, 2012, pp. 191–202. ISBN: 978-1-4503-1563-0. DOI: 10.1145/2384716.2384773. URL: http://doi.acm.org/10.1145/2384716.2384773.